Synchronous, Asynchronous and Hybrid Algorithms for DisCSPs*

Ismel Brito and Pedro Meseguer

Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{ismel|pedro}@iiia.csic.es

Abstract. There is some debate about the kind of algorithms that are most suitable to solve DisCSP. Synchronous algorithms exchange updated information with a low degree of parallelism. Asynchronous algorithms use less updated information with a higher parallelism. Hybrid algorithms combine both features. Lately, there is some evidence that synchronous algorithms could be more efficient than asynchronous ones for one problem class. In this paper, we present some improvements on existing synchronous and asynchronous algorithms, as well as a new hybrid algorithm. We provide an empirical investigation of these algorithms on n-queens and binary random DisCSPs.

1 Introduction

In the last years, the AI community has shown an increasing interest in distributed problem solving. Regarding distributed constraint reasoning, several synchronous and asynchronous backtracking procedures have been proposed to solve a constraint network distributed among several agents [15, 16, 6, 13, 1, 14, 4].

Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting [1]. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active one. These algorithms have a low degree of parallelism, but their agents receive updated information. In an asynchronous algorithm every agent is active at any time. They have a high degree of parallelism, but the information that any agent knows about other agents is less updated than in synchronous procedures.

There is some debate around the efficiency of these two type of algorithms. The general opinion was that asynchronous algorithms were more efficient than the synchronous ones, because of their higher concurrency [2]. In the last decade, attention was

* This research is supported by the REPLI project TIC-2002-04470-C03-03.

[1] Except for special topological arrangements of the constraint graph. See [3] for a synchronous algorithm where several agents are active concurrently.

[2] However, a careful reading of [17] shows that "synchronous backtracking might be as efficient as asynchronous backtracking due to the communication overhead" (footnote 15).

mainly devoted to the study and development of asynchronous procedures, which represented a new approach with respect to synchronous ones, directly derived from centralized algorithms.

Recently, Zivan and Meisels reported that the performance of a distributed and synchronous version of Conflict-Based Backjumping (*CBJ*) surpasses Asynchronous Backtracking (*ABT*) for the random problem class $\langle n = 10, m = 10, p_1 = 0.7 \rangle$.

In this paper we continue this line of research, and we study the performance of three different procedures, one synchronous, one asynchronous and one hybrid, for solving sparse, medium and dense DisCSPs. The synchronous algorithm is *SCBJ*, a distributed version of the Conflict-Based Backjumping (*CBJ*) [12] algorithm. The asynchronous algorithm is the standard *ABT* enhanced with some heuristics. The hybrid algorithm is *ABT-Hyb*, a novel *ABT*-like algorithm, where some synchronization is introduced to avoid redundant messages. In addition, we present a detailed approach for processing messages by packets instead of processing messages one by one, in *ABT* and *ABT-Hyb*. We also provide an experimental evaluation for new low-cost heuristics for variable and value reordering.

The rest of the paper is organized as follows. In Section 2 we recall some basic definitions of DisCSP. In Section 3 we recall two existing algorithms for DisCSP solving: the synchronous *SCBJ*, and the asynchronous *ABT*. In Section 4 we present *ABT-Hyb*, a new hybrid algorithm that combines asynchronous and synchronous elements, proving its soundness and completeness. In Section 5 we describe the experimental setting (including some implementation details) and discuss the experimental results. Finally, Section 6 contains several conclusions and directions of further work.

2 Distributed CSP

A constraint network is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of their respective finite domains, and $\mathcal{C}$ is a set of constraints declaring those value combinations which are acceptable for variables. The CSP involves finding values for the problem variables satisfying all constraints. We restrict our attention to constraints relating two variables, namely *binary* constraints. A constraint among the variables x_i and x_j will be denoted by c_{ij}.

A distributed CSP (DisCSP) is a CSP where the variables, domains and constraints of the underlying network are distributed among automated agents. Formally, a finite variable-based distributed constraint network is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before. $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents, and $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent. Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_{ij} is known by the agent owner of x_i and x_j, and it is unknown by the other agents. Usually, it is considered that an inter-agent constraint c_{ij} is known by the agents $\phi(x_i)$ and $\phi(x_j)$ [6, 17].

A solution of a distributed CSP is an assignment of values to variables satisfying every constraint (although distributed CSP literature focuses mainly on solving inter-agent constraints). Distributed CSPs are solved by the collective and coordinated action

of agents $\mathcal{A}$. Agents communicate by exchanging messages. It is assumed that the delay in delivering a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent.

For simplicity purposes, and to emphasize on distribution aspects, along the rest of the paper we assume that each agent owns exactly one variable. We identify the agent number with its variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). For this assumption, in the following we do not differentiate between a variable and its owner agent.

3 Existing Algorithms for DisCSP

3.1 Synchronous Search: SCBJ

Synchronous procedures can be directly derived from constraint algorithms in centralized search when extended to distributed environments. Generally, only one agent is active at any time in a a synchronous algorithm. Because of this, the active agent has always updated information, in the form of either a partial solution (from the part of the problem already assigned) or a backtracking.

The synchronous backtracking (*SBT*) algorithm for DisCSP was presented in [17]. Synchronous Conflict-Based Backjumping (*SCBJ*) [21] is a distributed version of the centralized Conflict-Based Backjumping (CBJ) algorithm [11]. While *SBT* performs chronological backtrackking, *SCBJ* does not. Each agent keeps the *conflict set* (*CS*), formed by the assigned variables which are inconsistent with some value of the agent variable. Let *self* be a generic agent. When a wipe-out occurs, it allows to *self* to backtrack directly to the closest conflict variable in CS_{self}, say x_i and sends $CS_{self} - \{x_i\}$ to be added to CS_i. Like *SBT*, *SCBJ* exchanges *Info* and *Back* messages, which are processed as follows (*self* is the receiver):

- *Info(partial-solution)*. *self* receives the partial solution, assigns its variable consistently, selects the next variable and sends the new partial solution to it in a *Info* message. If it has no consistent value, *self* sends a *Back* message to the closest variable in CS_{self}.
- *Back(conflict-set)*. *self* has to change its value, because *sender* has no value consistent with the partial solution. The current value of *self* is discarded, and the new conflict-set of *self* is the union of its old conflict-set and the one received. After this, *self* behaves as after receiving a *Info* message.

After receiving any of these messages, *self* becomes the active agent. *self* passes the privilege to other agent sending to it an *Info* or a *Back* message. The search ends unsuccessfully when any agent encounters an empty domain and its *CS* is empty. Otherwise, a solution will be found when the last agent is reached and there is a consistent value for it.

3.2 Asynchronous Search: ABT

In asynchronous search, all agents are active at any time, having a high degree of parallelism. Asynchronous Backtracking (*ABT*) [15, 17–19] was a pioneer asynchronous

algorithm to solve DisCSP. *ABT* requires a total agent ordering. Agent *i* has higher priority than *j* if *i* appears before *j* in the ordering. Each agent keeps its own agent view and nogood store. Considering a generic agent *self*, the agent view of *self* is the set of values that it believes to be assigned to its higher priority agents. The nogood store keeps nogoods as justifications of inconsistent values.

When *self* makes an assignment, it sends *Info* messages, to its lower priority agents, informing about its current assignment. When *self* receives a *Back* message, the included nogood is accepted if it is consistent with *self*'s agent view, otherwise it is discarded as obsolete. An accepted nogood is added to *self*'s nogood store to justify the deletion of the value it targets. In standard *ABT*, when *self* cannot take any value consistent with its agent view, because of the original constraints or because of the received nogoods, new nogoods are generated as inconsistent subsets of the agent view, and are sent, as *Back* messages, to the closest agent involved, causing backtracking.

In our *ABT* implementation, we keep a single nogood per removed value. When there is no value consistent with the agent view, a new nogood is generated by resolving all nogoods, as described in [1]. This nogood is sent in a *Back* message.

If *self* receives a nogood mentioning another agent not connected with it, *self* requires to add a link from that agent to *self*. *self* sends an assignment to that agent and after received, a link from the other agent to *self* will exist. The search terminates when achieving quiescence in the network, meaning that a solution has been found because all agents are agree with their current assignment, or when the empty nogood is generated, meaning that the problem is unsolvable.

4 Hybrid Search: ABT-Hyb

In *ABT*, many *Back* messages are obsolete when they arrive to the receiver. *ABT* could save much work if these messages were not sent. Although the sender agent cannot detect those messages that will become obsolete when reaching the receiver, it is possible to avoid sending those which are redundant.

Let *self* be a generic agent. When *self* sends a *Back* message, it performs a new assignment and informs of it to lower priority agents, without waiting to receive any message showing the effect of the *Back* message in higher agents. This can be a source of inefficiency in the following situation. If k sends a *Back* message to j causing a wipeout in j, then j sends a *Back* message to some previous agent i. If j takes the same value as before and sends an *Info* message to k before i changes its value, k will find again the same inconsistency so it will send the same nogood to j in a *Back* message. Agent j will discard this message as obsolete, sending again its value in an *Info*. The process is repeated generating useless messages, until some higher variable changes its value and the corresponding *Info* arrives to j and k.

Based on this intuition, we present *ABT-Hyb*, a hybrid algorithm that combines asynchronous and synchronous elements. *ABT-Hyb* behaves like *ABT* when no backtracking is performed: agents take their values asynchronously and inform lower priority agents. However, when an agent has to backtrack, it does it synchronously as follows. If *self* has no value consistent with its agent view and its nogood store, it sends a *Back* message and enters in a *waiting* state. In this state, *self* has no assigned value, and it

does not send out any message. Any received *Info* message is accepted, updating *self*'s agent view accordingly. Any received *Back* message is rejected as obsolete, since *self* has no value assigned. *self* leaves the waiting state when receiving one the following messages,

1. an *Info* message that allows *self* to has a value consistent with its agent view or,
2. an *Info* message from the receiver of the last *Back* message (the one causing to enter the waiting state) or,
3. a *Stop* message informing that the problem has not solution.

When *self* receives one of these messages, it leaves the waiting state. At this point, *ABT-Hyb* switches to *ABT*.

Like in *ABT*, the problem is unsolvable if during the search an empty nogood is derived. Otherwise, a solution is found when no messages are travelling through the network (i.e.quiescence is reached in the network). No matter the synchronous backtracking, *ABT-Hyb* inherits the good theoretical properties of *ABT*, namely soundness, completeness and termination. To proof these properties, we start with some lemmas.

Lemma 1. *In ABT-Hyb, no agent will stay forever in a waiting state.*

Proof. In *ABT-Hyb*, an agent enters the waiting state after sending a *Back* message to a higher priority agent. The first agent (x_1) in the ordering will not enter in the waiting state because no *Back* message departs from it. Suppose that no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, and suppose that x_k enters the waiting state after sending a *Back* message to x_j ($1 \leq j \leq k-1$). We will show that x_k will not be forever in the waiting state.

When x_j receives the *Back* message, there are two possible states:

1. x_j is waiting. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point. If x_j has a value consistent with its new agent view, it will send it to x_k in an *Info* message. If x_j has no value consistent with its new agent view, it will backtrack and enter again in a waiting state. This can be done a finite number of times (because there is a finite number of values per variable) before finding a consistent value or discovering that the problem has no solution generating a *Stop* message. In both cases, x_k will leave the waiting state.
2. x_j is not waiting. The *Back* message could be:
   (a) Obsolete in the value of x_j. In this case, there is an *Info* message travelling from x_j to x_k that has not arrived to x_k. After receiving such a message, x_k will leave the waiting state.
   (b) Obsolete not in the value of x_j. In this case, x_j resends to x_k its value by an *Info* message. After receiving such a message, x_k will leave the waiting state.
   (c) Not obsolete. The value of x_j is forbidden by the nogood in the *Back* message, and a new value is tried. If x_j finds another value consistent with its agent view, it takes it and send an *Info* message to x_k, which will leave the waiting state. Otherwise, x_j has to backtrack to a previous agent in the ordering, and enters the waiting state. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point, and as explained in the point 1 above, it will cause that x_k will leave the waiting state as well.

Therefore, we conclude that x_k will not stay forever in the waiting state. □

Lemma 2. *In ABT-Hyb, if an agent is in a waiting state, the network is not quiescent.*

Proof. An agent is in a waiting state after sending a *Back* message. Because Lemma 1, this agent will leave the waiting state in finite time. This is done after receiving an *Info* or *Stop* message. Therefore, if there is an agent in a waiting state, the network cannot be quiescent at least until one of those messages has been produced. □

Lemma 3. *A nogood, discarded as obsolete because the receiver is in a waiting state, will be resent to the receiver until the sender realizes that it has been solved, or the empty nogood has been derived.*

Proof. If an agent k sends a nogood to an agent j that is in a waiting state, this nogood is discarded and agent k enters the waiting state. From Lemma 1, no agent can stay forever in a waiting state, so agent k will leave that state in finite time. This is done after receiving either,

1. An *Info* message from j. If this message does not solve the nogood, it will be generated and resend to j. If it solves it, this nogood is not generated, exactly in the same way as *ABT* does.
2. An *Info* message allowing a consistent value for k. In this case, the nogood is solved, so it is not resent again.
3. A *Stop* message. The process terminates without solution.

Therefore, we conclude that the nogood is sent again until it is solved (either by an *Info* message from j or from another agent) or the empty nogood is generated. □

Proposition 1. *ABT-Hyb is sound.*

Proof. From Lemma 2, *ABT-Hyb* reaches quiescence only when no agent is in a waiting state. From this fact, *ABT-Hyb* soundness derives directly from *ABT* soundness: when the network is quiescent all agents satisfy their constraints, so the current assignments of agents form a solution. If this would not be the case, at least one agent would detect a violated constraint and it would send a message, breaking the quiescence assumption. □

Proposition 2. *ABT-Hyb is complete and terminates.*

Proof. From Lemma 3, the synchronicity of backtracking in *ABT-Hyb* does not cause to ignore any nogood. Then, *ABT-Hyb* explores the search space as good as *ABT* does. From this fact, *ABT-Hyb* completeness comes directly from *ABT* completeness. New nogoods are generated by logical inference from the initial constraints, so the empty nogood cannot be derived if there is a solution. Total agent ordering causes that backtracking discards one value in the highest variable reached by the *Back* message. Since the number of values is finite, the process will find a solution if it exists, or it will derive the empty nogood otherwise.

To see that *ABT-Hyb* terminates, we have to prove that no agent falls into an infinite loop. This comes from the fact that agents cannot stay forever in the waiting state (Lemma 1), and that *ABT* agents cannot be in an endless loop. □

Alternatively to synchronous backtracking, we can avoid resending redundant *Back* messages assuming exponential-space algorithms. Let assume that *self* stores every nogood sent, while it is not obsolete. If a wipe-out occurs in *self*, if the new generated nogood is equal to one of the stored nogoods, it is not sent. This allows *self* not sending identical nogoods until some higher agent changes its value and the corresponding *Info* arrives to *self*. But it requires exponential space, since the number of nogoods generated could be exponential in the number of agents with higher priority than *self*. A similar idea is also found in [16] for the asynchronous weak-commitment algorithm (*AWC*).

5 Experimental Results

We have tested *SCBJ*, *ABT* and *ABT-Hyb* algorithms on the distributed n-queens problem and on random binary problems. Algorithmic performance is evaluated considering computation and communication costs. In synchronous algorithms, the computation effort is measured by the total number of constraint checks (cc), and the global communication effort is evaluated by the total number of messages exchanged among agents (msg).

For the asynchronous algorithms *ABT* and *ABT-Hyb*, computation effort is measured by the number of "concurrent constraint checks" (ccc), which was defined in [8], following Lamport's logic clocks [10]. Each agent has a counter for its own number of constraint checks. The number of concurrent constraint checks is computed by attaching to every message the current counter of the constraint checks of the sending agent. When an agent receives a message, it updates its counter to the higher value between its own counter and the counter attached to the received message. When the algorithm terminates, the highest value among all the agent counters is taken as the number of concurrent constraint checks. Informally, this number approximates the longest sequence of constraint checks not performed concurrently. As for synchronous search, we evaluate the global communication effort as the total number of messages exchanged among agents (msg).

5.1 Implementation Details

Nogood management. To assure polynomial space in *ABT* and *ABT-Hyb*, we keep one nogood per forbidden value. However, if several nogoods are available for each value, it may be advisable to choose the most appropriate resolvent in order to speed up search. With this aim, we implement the following heuristic. If a value is forbidden for some stored nogood, and a new nogood forbidding the same value arrives, we store the nogood with the highest possible lowest variable involved. Notice that, even those nogoods which are obsolete on the value of the receiving variable can be used to select the most suitable nogood with respect to the heuristic.

Saving messages. In asynchronous algorithms, some tricks can be used to decrease the number of messages exchanged. We implement the following:

1. *Value in* AddL. When a new link with agent k is requested by *self*, instead of sending the *AddL* message and assuming this assignment until a confirmation is received, *ABT* include in the *AddL* message the value of x_k recorded in the received nogood. After reception of the *AddL* message, agent k informs *self* of its current value only if it is different from the value contained in the *AddL* message. In this way, some messages may be saved.
2. *Avoid resending same values*. *ABT* can keep track of the last value taken by *self*. When selecting a new value, if it happens that the new value is the same as the last value, *self* does not resend it to $\Gamma^+(self)$, because this information is already known. Again, this may save some messages.

Processing Messages by Packets. *ABT* agents can process messages one by one, reacting as soon as a message is received. However, this strategy of *single-message process* may cause some useless work. For instance, consider the reception of an *Info* message reporting a change of an agent value, immediately followed by another *Info* from the same agent. Processing the first message causes some work that becomes useless as soon as the second message arrives. More complex examples can be devised, causing to waste substantial effort.

To prevent useless work, instead of reacting after each received message, the algorithm reads all messages that are in the input buffer and stores them in internal data structures. Then, the algorithm processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message. We call this strategy *processing messages by packets*, where a packet is the set of messages that are read from the input buffer until it becomes empty. Somehow, this idea was mentioned in [17] and [21]. In the latter, a comparison between *single-message process* and *processing messages by packets* is presented. However, in none of them a formal protocol for *processing messages by packets* is completely developed.

When an agent processes messages by packets, it reads all messages from its input buffer, and processes them as a whole. The agent looks for any consistent value after its agent view and its nogood store are updated with these incoming messages. To do that, we propose a protocol which requires three lists to store the incoming messages, the *Info*-$List$, *Back*-$List$ and the *AddL*-$List$. In each list is stored the messages of the corresponding type, following the reception order. Each list of messages is processed as follows.

1. *Info*-$List$. First, the *Info*-$List$ is processed. For each sender agent, all *Info* messages but the last are ignored. The remaining *Info* messages update *self* agent view, removing nogoods if needed.
2. *Back*-$List$. Second, the *Back*-$List$ is processed. Obsolete *Back* messages are ignored. *self* stores nogoods of no obsolete messages, and it sends *AddL* messages to unrelated agents appearing in those nogoods. For those messages containing the correct current value of *self*, the sender is recorded in *RemainderSet*.
3. *AddL*-$List$. Third, the *AddL*-$List$ is processed updating $\Gamma^+(self)$ without sending the *Info* message.

lex	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	1,612	170	2,223	740	1,699	502
15	31,761	2,231	56,412	13,978	32,373	6,881
20	6,518,652	306,337	11,084,012	2,198,304	6,086,376	995,902
25	1,771,192	70,336	3,868,136	693,832	1,660,448	271,092
rand	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	965	91	1,742	332	916	238
15	4,120	247	7,697	1,185	4,007	786
20	19,532	921	20,661	4,772	15,720	2,748
25	21,372	746	31,849	6,553	27,055	3,863
min	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	2,800	204	3,716	896	2,988	555
15	35,339	2,210	49,442	11,055	32,303	5,906
20	215,816	10,765	320,278	63,378	165,338	28,686
25	19,949,074	791,089	38,450,786	6,716,505	17,614,330	2,795,319

Table 1. Results for distributed n-queens with lex, random and min-conflict value ordering.

4. Consistent value. Fourth, *self* tries to find a value consistency with the agent view. If a wipe-out happens in this process, the corresponding *Back* message is sent, and a consistent value is searched.
5. *Info* sent. Fifth, *Info* messages containing *self* current value are sent to all agents in $\Gamma^+(self)$ and to all agents in *RemainderSet*. The three lists become empty.

As described in Section 3.2, the search ends when quiescence is reached (i.e. all agents are happy with their current assignment) or an empty nogood is derived.

5.2 Distributed n-queens Problem

The distributed n-queens problem is the classical n-queens problem (locate n queens in an $n \times n$ chessboard such that no pair of queens are attacking each other) where each queen is hold by an independent agent. We have evaluated the algorithms for four dimensions $n = 10, 15, 20, 25$. In Table 1 we show the results in terms of constraint checks/concurrent constraint checks and total number of messages exchanged, averaged over 100 executions with different random seeds (ties are broken randomly). Lexicographic (static) variable ordering has been used for *SCBJ*, *ABT*, and *ABT-Hyb*. Three value ordering heuristics have been tested *lex* (lexicographic), *rand* (random) and *min* (min-conflicts) [9] on all the algorithms. Given that an exact *min* computation requires extra messages, we have made an approximation, which consists of computing the heuristic assuming initial domains. With this approximation, the *min* value ordering heuristic can be computed in a preprocessing step.

We observe that the random value ordering provides the best performance for every algorithm and every dimension tested. Because of that, in the following we concentrate our analysis on the results of random value ordering.

Considering the relative performance of asynchronous algorithms, *ABT-Hyb* is always better than *ABT*, in both number of concurrent constraint checks and total number of messages. It is relevant to scrutinize the improvement of *ABT-Hyb* over *ABT* with respect to the type of messages. In Table 2, we provide the total number of messages per message type for *SCBJ*, *ABT* and *ABT-Hyb* with random value ordering. In *ABT-Hyb* the number of obsolete *Back* messages decreases in one order of magnitude with respect the same type of messages in *ABT*, causing *ABT-Hyb* to improve over *ABT*. However, this improvement goes beyond the savings in obsolete *Back* messages, because *Info* and *Back* messages decrement to a larger extent. This is due to the following collective effect. When an *ABT* agent sends a *Back* message, it tries to get a new consistent value without knowing the effect that backtracking causes in higher priority agents. If it finds such a consistent value, it informs to lower priority agents using *Info* messages. If it happens that this value is not consistent with new values that backtracking causes in higher priority agents, these *Info* messages would be useless, and new *Back* messages would be generated. *ABT-Hyb* tries to avoid this situation. When an *ABT-Hyb* agent sends a *Back* message, it waits until it receives notice of the effect of backtracking in higher priority agents. When it leaves the waiting state, it tries to get a new consistent value. At this point, it knows some effect of the backtracking on higher priority agents, so the new value will be consistent with it. In this way, the new value has more chance to be consistent with all higher priority agents, and the *Info* messages carrying it will be more likely to make useful work.

Considering the performance of synchronous vs. asynchronous algorithms, we compare *SCBJ* against *ABT-Hyb* with random value ordering. In terms of computation effort (constraint checks) *SCBJ* performs better than *ABT-Hyb* for $n = 25$ and worse for $n = 20$, with very similar results for $n = 10, 15$. In terms of communication cost, *SCBJ* uses less messages than *ABT-Hyb* for the four dimensions tested. This comparison should be qualified, noting that the lenght of *Info* messages differ from synchronous to asynchronous algorithms. In *SCBJ*, an *Info* message contains the partial solution which could be of size n, while in *ABT-Hyb* an *Info* message contains a single assignment of size 1. Assuming that the communication cost depends more crucially on the number of messages than on their length, we conclude that *SCBJ* is more efficient in communication terms than *ABT-Hyb*. Considering both aspects, computation effort and communication cost, *SCBJ* seems to be the algorithm of choice for the n-queens problem.

5.3 Random Problems

Uniform binary random CSPs are characterized by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the number of values per variable, p_1 the network *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. We have tested random instances of 16 agents and 8 values per agent, considering three connectivity classes, sparse (p_1=0.2), medium (p_1=0.5) and dense (p_1=0.8).

rand	SCBJ		ABT			ABT-Hyb		
n	Info	Back	Info	Back	Obsol	Info	Back	Obsol
10	55	36	251	81	24	195	43	2
15	146	101	901	284	91	649	137	10
20	539	382	3,612	1,160	408	2,293	455	38
25	452	294	5,027	1,526	520	3,240	623	50

Table 2. Number of messages exchanged by *SCBJ*, *ABT* and *ABT-Hyb* per message type, for the distributed n-queens problem with random value ordering.

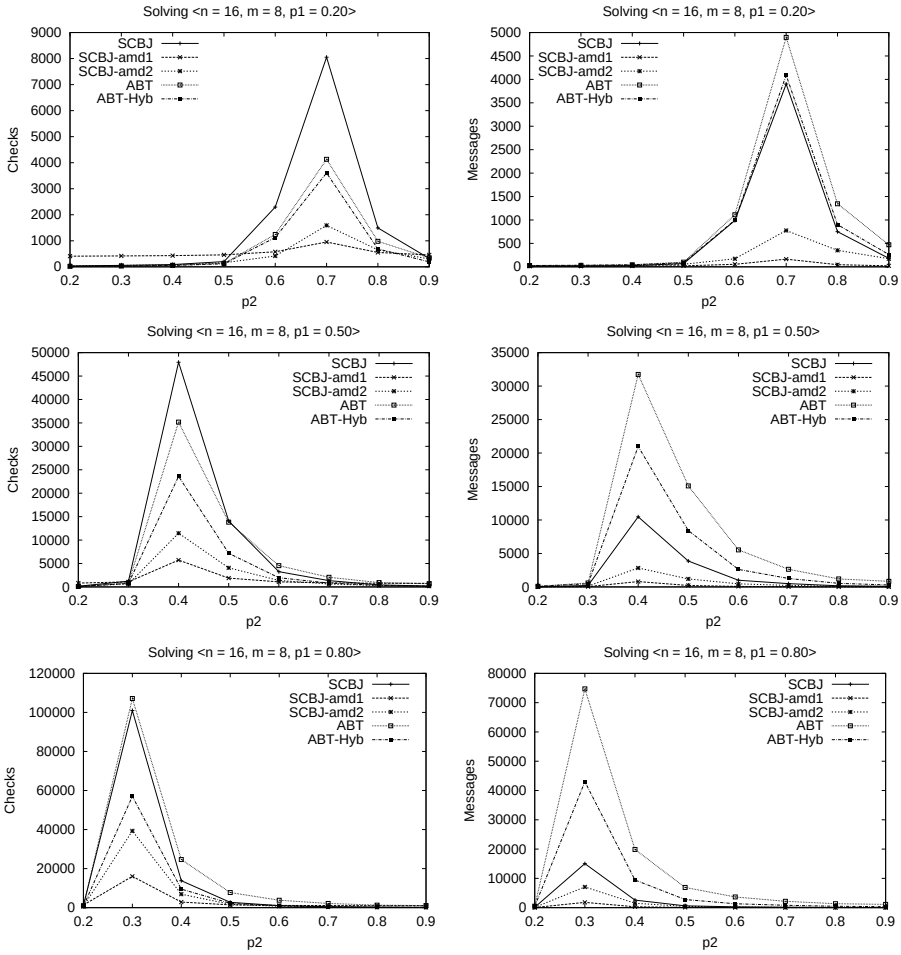

Fig. 1. Constraint checks and number of messages for *SCBJ*, *SCBJ-amd1*, *SCBJ-amd2*, *ABT* and *ABT-Hyb* on binary random problems.

In a synchronous algorithm, it is simple to implement some heuristic for dynamic variable ordering. Considering the heuristic of minimum domain, an exact computation

rand	SCBJ		SCBJ-amd1		ABT				ABT-Hyb			
p_2	Info	Back	Info	Back	Info	Back	Obsol	Link	Info	Back	Obsol	Link
0.20	2,647	1,254	100	63	3,587	1,310	320	26	3,141	949	53	24
0.50	6,913	3,556	477	321	24,725	7,025	2,336	40	17,650	3,335	321	37
0.80	9,761	5,265	1,052	758	58,283	16,432	6,497	19	37,046	5,956	755	18

Table 3. Number of messages exchanged by *SCBJ*, *SCBJ-amd1*, *ABT* and *ABT-Hyb* per message type, for random binary problems with random value ordering.

requires extra messages. To avoid this, we have implemented the following approximations,

- AMD1. Each agent computes the interval $[min_i, max_i]$ of the minimum and maximum number of inconsistent values in the domain of every unassigned variable x_i with the partial solution. This interval is included in the *Info* message. Then, the next variable to be assigned is chosen as follows: (i) if there is x_i such that $min_i \geq min\{d, max_j\}, \forall x_j$ unassigned, selects x_i (where d is the domain size); (ii) otherwise, selects the variable with maximum max_j.
- AMD2. This approach only computes the current domains of the unassigned variables after *Back* messages. When *self* sends a *Back* message to x_j, instead of sending it directly to x_j it goes chronologically. Each intermediate variable recognizes that it is not its destination, and it includes the current size of its domain in the message. This messages ends in x_j and after assigning it, the minimum domain heuristic without considering the effect of x_j's assignment can be applied on the subset of intermediate variables. It causes some extra messages, but its benefits pay-off.

In Figure 1, we report results averaged over 100 executions for *SCBJ*, *SCBJ-amd1*, *SCBJ-amd2*, *ABT* and *ABT-Hyb*, with random value ordering.

Considering synchronous algorithms, approximating minimum domains heuristic is always beneficial both in computation effort and in communication cost. Consistently in the three classes tested, the approximation *amd1* provides better results than *amd2*, both in terms of checks and messages. When using *amd1*, the baseline of constraint checks is not zero, due to the heuristic computation done as a preprocessing step.

Considering asynchronous algorithms, we observe again that *ABT-Hyb* is always better than *ABT* for the three problem classes, in both computation effort and communication cost. We believe that this is due to the effect already described for the distributed n-queens problem. This is confirmed after analyzing the number of messages per message type of Table 3.

Comparing the performance of synchronous vs. *ABT-Hyb*, we observe the following. In terms of computation effort (constraint checks), *SCBJ* is always worse than *ABT-Hyb*, and *SCBJ* is often the worst algorithm (except in the $\langle 16, 8, 0.8 \rangle$ class, where it is the second worst). This behaviour changes dramatically when adding the minimum domain heuristic approximations: *SCBJ-amd1* and *SCBJ-amd2* are the best and second best algorithms in the three classes tested, and they are always better than *ABT-Hyb*.

min	SCBJ		SCBJ-amd1		SCBJ-amd2		ABT		ABT-Hyb	
p_1	cc	msg	cc	msg	cc	msg	ccc	msg	ccc	msg
0.20	7,100	3,277	907	153	1,811	687	3,771	4,006	3,448	3,535
0.50	44,024	9,367	5,637	783	11,677	2,669	30,719	26,840	22,227	19,141
0.80	102,153	15,111	16,206	1,843	40,449	7,142	101,492	70,033	58,428	43,459

Table 4. Results near of the pick of difficulty on binary random classes $\langle n = 16, m = 8 \rangle$ with min-conflict value ordering.

Regarding communication costs, synchronous algorithms are always better than asynchronous ones: consistently in the three classes tested, *SCBJ-amd1*, *SCBJ-amd2* and *SCBJ* are the three best algorithms (in this order). Again, the addition of minimum domain approximations is very beneficial. As mentioned in Section 5.2, $Info$ messages are of different sizes in synchronous and asynchronous algorithms. Under the same assumptions (communication costs depends more on the number of messages exchanged than on their length), we conclude that for solving random binary problems, *SCBJ-amd1* is the algorithm of choice.

We have also tested the three problem classes using the min-conflict value ordering. Results appear in Table 4 for the peak of maximum difficulty. We observe a minor but consistent improvement of all the algorithms with respect to the random value ordering. In this case, the relative ranking of algorithms obtained with random value ordering remains, *SCBJ-amd1* being the algorithm with the best performance.

We have also tested *ABT* and *ABT-Hyb* with random message delays. This issue was raised first in [5], and subsequently in [21]. Preliminary results show that *ABT* decreases performance and also *ABT-Hyb* does, but to a lesser extent. This last algorithm exhibits a more robust behavior in presence of random delays. It is worth noting that synchronous algorithms do not increase the number of checks or messages in presence of delays.

6 Conclusions

We have presented three algorithms, one synchronous *SCBJ*, one asynchronous *ABT* and one hybrid *ABT-Hyb*, the two first being already known. We have proposed *ABT-Hyb*, a new algorithm that combines asynchronous and synchronous elements. *ABT-Hyb* can be seen as an *ABT*-like algorithm where backtracking is synchronized: an agent that initiates backtracking cannot take a new value before having some notice of the effect of its backtracking. This causes a kind of "contention effect" in backtracking agents. Their decisions tend to be better founded than the corresponding decisions taken by *ABT* agents, and therefore they are more likely to succeed. *ABT-Hyb* inherits the good theoretical properties of *ABT*: it is sound, complete and terminates.

We have implemented *ABT* and *ABT-Hyb* with a strategy for processing messages by packets, together with some simple ideas to improve performance. On *SCBJ* we have proposed two approximations for the minimum domain heuristic. Empirically we have observed that *ABT-Hyb* clearly improves over *ABT*, in both computation effort and communication costs. Comparing *SCBJ* with *ABT-Hyb*, we observe that *SCBJ* always requires less messages than *ABT-Hyb*, for both problems tested. Considering computation effort, *SCBJ* requires a similar effort as *ABT-Hyb* in distributed n-queens, while

SCBJ requires more effort than *ABT-Hyb* for binary random problems. However, when enhanced with minimum domain approximation for dynamic variable ordering, *SCBJ-amd1* is the best algorithm in terms computation effort and in number of messages exchanged. Grouping these evidences together, we conclude that synchonous algorithms enhanced with some minimum domain approximation are globally more efficient than asynchronous ones. This does not mean that synchronous algorithms should always be preferred to asynchronous ones, since they offer different functionalities (synchronous algorithms are less robust to network failures, privacy issues are not considered, etc.). But for applications where efficiency is the main concern, synchronous algorithms seems to be quite good candidates to solve DisCSP.

References

1. Bessière C., Maestre A. and Meseguer P. Distributed Dynamic Backtracking. *IJCAI-01 Workshop on Distributed Constraint Reasoning*, 9-16, Seattle, USA, 2001.
2. Bitner J. and Reingold E. Backtrack programming techniques. *Communications of the ACM*, **18**:11, 651–656, 1975.
3. Collin Z., Dechter R., Shmuel K. On the Feasibility of Distributed Constraint Satisfaction. *In Proc. of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, 318–324, 1991.
4. Dechter R. and Pearl J. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, **34**, 1–38, (1988).
5. Fernandez C., Bejar R., Krishnamachari Gomes, K. Communication and Computation in Distributed CSP Algorithms. *In Proc. Principles and Practice of Constraint Satisfaction Programming (CP-2002)*, 664–679, Ithaca NY, USA, July, 2002.
6. Hamadi Y., Bessière C., Quinqueton J. Backtracking in Distributed Constraint Networks. *In Proc. of the 13th ECAI*, 219–223, Brighton, UK, 1998.
7. Hirayama K. and Yokoo M. The Effect of Nogood Learning in Distributed Constraint Satisfaction. *In Proceedings ICDCS'00*, 169–177. 2000
8. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS-02 Workshop on Distributed Constraint Reasoning*, 86–93, Bologna, Italy, 2002.
9. Minton S. and Johnston M. and Philips A. and Laird P. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, **58**, 161–205, 1992.
10. Lamport L. Time, Clock, and the Ordering of Evens in a Distributed System. *Communications of the ACM*, **21(7)**, 558–565, 1978.
11. Prosser, P. Hybrid Algorithm for the Constraint Satisfaction Problem. *Computational Intelligence*, **9**, 268–299, 1993.
12. Prosser, P. Domain Filtering can Degrade Intelligent Backtracking Search. *Proc. IJCAI*, 262–267, 1993.
13. Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous Search with Aggregations. *In Proc. of the 17th AAAI*, 917–922, 2000.
14. Silaghi M.C., Sam-Haroud D., Faltings B. *Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering*. Tech. Report EPFL, 2001.
15. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *In Proc. of the 12th International Conference on Distributed Computing System*, 614–621, 1992.

16. Yokoo M. Asynchronous Weak-commitment Search for Solving Distributed Constraints Satisfaction Problems. *In Proceeding of the First International Conference on Principles and Practice of Constraint Programming (CP-1995)* 88–102, 1995.
17. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10**, 673–685, 1998.
18. Yokoo M., Ishida T. Search Algorithms for Agents. In *Multiagent Systems*, G. Weiss editor, Springer, 1999.
19. Yokoo M. *Distributed Constraint Satisfaction*, Springer, 2001.
20. Yokoo M. , Suzuki, Hirayama K. Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information. *In Proc. of the 8th CP*, 387–401, 2002.
21. Zivan, R. and Meisels, A. *Synchronous and Asynchronous Search on DisCSPs.* In Proc. of EUMAS-2003, Oxford, UK, 2003