Constraints (2007) 12:131–150
DOI 10.1007/s10601-006-9013-5

Asynchronous Forward-checking for DisCSPs

Amnon Meisels · Roie Zivan

Published online: 12 January 2007

Abstract A new search algorithm for solving distributed constraint satisfaction problems (*DisCSP*s) is presented. Agents assign variables sequentially, but perform forward checking asynchronously. The asynchronous forward-checking algorithm (*AFC*) is a distributed search algorithm that keeps one consistent partial assignment at all times. Forward checking is performed by sending copies of the partial assignment to all unassigned agents concurrently. The algorithm is described in detail and its correctness proven. The sequential assignment method of *AFC* leads naturally to dynamic ordering of agents during search. Several ordering heuristics are presented. The three best heuristics are evaluated and shown to improve the performance of *AFC* with static order by a large factor. An experimental comparison of AFC to asynchronous backtracking (*ABT*) on randomly generated DisCSPs is also presented. *AFC* with ordering heuristics outperforms *ABT* by a large factor on the harder instances of random DisCSPs. These results hold for two measures of performance: number of non-concurrent constraints checks and number of messages sent.

Keywords Distributed CSPs · Asynchronous search · Forward-checking

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

A. Meisels (✉) · R. Zivan
Department of Computer Science, Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
e-mail: am@cs.bgu.ac.il

R. Zivan
e-mail: zivanr@cs.bgu.ac.il

1 Introduction

Distributed constraints 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 [19, 20]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them, to check consistency of their proposed assignments against constraints among variables that belong to different agents [1, 20].

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 time s of different wards and generates a complex Distributed CSP [19].

Several asynchronous backtracking algorithms for *DisCSP*s have been proposed in recent years [2, 17, 20]. All of these algorithms process assignments of agents asynchronously and rely on Nogoods for their correctness and termination. In asynchronous backtracking, agents perform assignments asynchronously and send out messages to constraining agents, informing them about their assignments [2, 20]. Due to the asynchronous nature of agents' operations, the global assignment state at any particular instance during the run of an asynchronous backtracking algorithm is in general inconsistent.

The present paper proposes a new distributed search algorithm on *DisCSP*s, *Asynchronous Forward-Checking (AFC)*. The *AFC* algorithm processes only consistent partial assignments and processes assignments synchronously. The innovation of the proposed algorithm lies in processing forward checking (FC) asynchronously, hence its name *AFC*. In the proposed *AFC* algorithm, the state of the search process is represented by a data structure called *Current Partial Assignment (CPA)*. A *CPA* starts empty at some initializing agent that records its assignments on it and sends it to the next agent.

Each receiving agent adds its assignment to the *CPA*, if a consistent assignment can be found. Otherwise, it backtracks by sending the same *CPA* to a former agent to revise its assignment on the *CPA*. Each agent that performs an assignment on a *CPA* sends forward a copy of the updated *CPA*, requesting all agents to perform forward-checking. Agents that receive copies of assignments filter their domains and in case of a dead-end send back a *Not_OK* message. The concurrency of the *AFC* algorithm is achieved by the fact that forward-checking is performed concurrently by all agents. It is important to note that *AFC* performs forward checking against consistent partial assignments, using copies of *CPA*s.[1]

[1] AFC should not be confused with Distributed Forward Checking [12] which is a method for keeping agents assignments private in asynchronous backtracking.

The *AFC* algorithm includes a protocol that enables agents to process forward checking (FC) messages concurrently and yet block the assignment process at the agent that violates consistency with future variables. The sequential way in which agents extend a consistent partial assignment, makes dynamic ordering of agents as straightforward as in synchronous algorithms. While the best heuristic for the *AFC* algorithm requires additional messages to be exchanged between agents, a heuristic inspired by dynamic backtracking [6], which does not need any additional messages, was also found to be very effective (See Section 8).

The synchronous method of performing assignments in *AFC* may generate some confusion with simple synchronous backtracking algorithms for *DisCSP*s (*Synchronous Backtrack* [20], *CBJ* [23]). Synchronous Backtrack is the simplest *DisCSP* search algorithm and performs assignments sequentially and synchronously, one agent at a time in a fixed order. The proposed *AFC* algorithm performs assignments by one agent at a time, but *checks for consistency in an asynchronous process*. As will be evident, *AFC* is more efficient computationally than the best version of asynchronous backtracking (Section 8).

The *AFC* algorithm is described in detail in Section 3 and its correctness is proven in Section 4. Different ordering heuristics which can be used by the *AFC* algorithm are presented in Section 5. Section 7 presents a discussion of the privacy level of *AFC* and how it may be increased if the privacy requirements are higher. The impact of using the three best heuristics is evaluated in Section 8. The performance of *AFC* is compared to that of asynchronous backtracking (*ABT*) on randomly generated *DisCSP*s. *AFC* outperforms *ABT* by a large factor on the harder instances of random problems. This is true for all three measures of performance: the number of concurrent constraints checks, the number of concurrent steps of computation and the total number of messages sent (see Section 8). A discussion of the differences of the *AFC* algorithm from asynchronous backtracking and of its improved performance is presented in Section 9. Our conclusions are in Section 10.

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 [19, 20]. 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 t o 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.

Following all former work on *DisCSP*s, agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. An agent can send messages to any one of the other agents. The delay in delivering a message is assumed to be finite [20]. One simple form of messages for checking constraints, that appear in many distributed search algorithms, is to send a proposed assignment $< var, val >$, of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of its variables and returns a message that either acknowledges or rejects the proposed assignment (cf. [1, 20]).

3 Asynchronous Forward Checking (*AFC*)

The *AFC* algorithm combines the advantage of assigning values consistent with all former assignments and of propagating the assignments forward asynchronously. Assignments in *AFC* are performed by one agent at a time. The assigning agent keeps the partial assignment consistent. Each such assignment is checked by multiple agents concurrently. Although forward-checking is performed asynchronously, at most one backtrack operation is generated for a failure in a future variable.

Agents assign their variables only when they hold the current partial assignment (*CPA*). The *CPA* is a unique message that is passed between agents, and carries the partial assignment that agents attempt to extend into a complete solution by assigning their variables on it.

Forward checking is performed as follows. Every agent that sends the *CPA* forward sends copies of the *CPA*, in messages we term *FC_CPA*, to all agents whose assignments are not yet on the *CPA* (except for the agent the *CPA* itself is sent to). Agents that receive *FC_CPA*s update their variables domains, removing all values that conflict with assignments on the *FC_CPA*. Asynchronous forward checking enables agents an early detection of inconsistent partial assignments and initiates backtracks as early as possible. An agent that generates an empty domain as a result of a forward-checking operation, initiates a backtrack procedure by sending *Not_OK* messages which carry the inconsistent partial assignment which caused the empty domain. A *Not_OK* message is sent to all agents with unassigned variables on the (inconsistent) *CPA*. An agent that receives the *CPA* and is holding a *Not_OK* message, sends the *CPA* back in a backtrack message. The uniqueness of the *CPA* ensures that only a single backtrack is initialized, even for multiple *Not_OK* messages. In other words, when multiple agents reject a given assignment by sending *Not_OK* messages, *only one agent that received any of those messages will eventually backtrack*. The first agent that will receive a *CPA* and is holding a relevant *Not_OK* message. The *Not_OK* message becomes obsolete when the partial assignment it carries is no longer a subset of the *CPA*. (Other options for initializing backtrack operations were suggested by [13] see Section 6).

The *AFC* algorithm is run on each of the agents in the *DisCSP* and uses the following objects and messages:

– *CPA (current partial assignment)*: a message that carries the currently valid (and consistent) partial assignment. A *CPA* is composed of triplets of the form $< A, X, V >$ where A is the agent that owns variable X and V is the value that

was assigned to X by A. Each CPA contains a counter that is updated by each agent that assigns its variables on the CPA. This counter is used as a time-stamp by the agents in the AFC algorithm and is termed the Step-Counter ('SC'). The partial assignment in a CPA is maintained in the order the assignments were made by the agents.

- FC_CPA: a message that is an exact copy of a CPA. Every agent that assigns its variables on a CPA, creates an exact copy in the form of a FC_CPA (with the same SC) and sends it forward to all unassigned agents.
- Not_OK: agents update their domains whenever they receive FC_CPA messages. When an agent encounters an empty domain, during this process, it sends a Not_OK message. The Not_OK message carries *the shortest inconsistent subset of assignments* from the FC_CPA and informs other agents that this partial assignment is inconsistent with the sending agent's domain.
- $AgentView$: each agent holds a list of assignments which are its updated view of the current assignment state of all other agents. The $AgentView$ contains a consistency flag $AgentView.consistent$, that represents whether the partial assignment it holds is consistent. The $AgentView$ contains a $step\_counter(SC)$ which holds the value of the highest SC received by the agent.
- *Backtrack*: an inconsistent CPA (i.e. a 'Nogood') sent to the agent with the most recent conflicting assignment.

3.1 Algorithm Description

The main function of the algorithm AFC is presented in Fig. 1 and performs two tasks. If it is run by the initializing agent (IA), it initiates the search by generating a CPA (with $SC = 0$), and then calling function ***assign_CPA*** (line 2–4). All agents performing the main function wait for messages, and call the functions dealing with the relevant type of message received. The two functions dealing with receiving the CPA and assigning variables on it are presented in Fig. 1.

Function ***receive_CPA*** is called when the CPA is received either in a forward move or in a backtrack message. After storing the CPA, the agent checks its $AgentView$ status. If it is not consistent and it is a subset of the received CPA, this means that a backtrack of the CPA has to be performed. If the inconsistent $AgentView$ is not a subset of the received CPA, the CPA is stored as the updated $AgentView$ and it is marked consistent. This reflects the fact that the received CPA has revised assignments that caused the original inconsistency. The rest of the function calls $assign\_CPA$, to extend the current partial assignment. If the CPA is a backtrack, the last assignment is removed first (lines 8, 9). Otherwise, the $AgentView$ is updated to the received CPA and its consistency with current domains is checked and updated. The assignment of variables of the agent currently holding the CPA is performed by the function $assign\_CPA$.

Function ***assign_CPA*** tries to find an assignment for the agent's local variables, which is consistent with local constraints and does not conflict with previous assignments on the CPA. If the agent succeeds it sends forward the CPA or reports a solution, when the CPA includes all agents assignments (lines 2–5). If the agent fails to find a consistent assignment, it calls function *backtrack* after updating its $AgentView$ with the inconsistent partial assignment, that was just discovered (lines

```
AFC:
1.  done ← false
2.  if(IA)
3.    CPA ← generate_CPA
4.    assign_CPA
5.  while(not done)
6.    msg ← receive_msg
7.    switch msg.type
8.      stop: done ← true
9.      FC_CPA: forward_check
10.     Not_OK: process_Not_OK
11.     CPA: receive_CPA
12.     backtrack_CPA: receive_CPA

receive_CPA:
1.  CPA ← msg_CPA
2.  if(not AgentView.consistent)
3.    if(contains(CPA, AgentView))
4.      backtrack
5.    else
6.      AgentView.consistent ← true
7.  if(AgentView.consistent)
8.    if(msg.type = backtrack_CPA)
9.      remove_last_assignment
10.     assign_CPA
11.   else
12.     if(update_AgentView(CPA))
13.       assign_CPA
14.     else
15.       backtrack

assign_CPA:
1.  CPA ← add_local_assignments
2.  if(is_assigned(CPA))
3.    if(is_full(CPA))
4.      report_solution
5.      stop
6.    else
7.      CPA.SC++
8.      send(CPA,next)
9.      send(FC_CPA,other_unassigned_agents)
10. else
11.   AgentView ← shortest_inconsistent_partial_assignment
12.   backtrack
```

Fig. 1 AFC algorithm: receive and assign CPA

11–12). Whenever an agent sends forward a CPA (line 8), it sends a copy of it in a FC_CPA message to every other agent whose assignments are not yet on the CPA (line 9).

The rest of the AFC algorithm deals with backward moving CPAs and with propagation of the current assignment and is presented in Fig. 2.

Function ***backtrack*** is called when the agent is holding the CPA in one of two cases. Either the agent cannot find a consistent assignment for its variables, or its $AgentView$ is inconsistent and is found to be relevant with the received CPA. In case the agent is the IA the search ends unsuccessfully (lines 1–3). Other agents performing a backtrack operation, copy to the CPA the shortest inconsistent partial assignment, from their $AgentView$ (line 6), and send it back to the agent which is the owner of the last variable in that partial assignment. The $AgentView$ of the sending agent retains the Nogood that was sent back.

backtrack:
1. **if**(IA)
2. send(stop, all_other_agents)
3. done ← true
4. **else**
5. AgentView.consistent ← false
5. backTo ← last(AgentView)
6. CPA ← AgentView
7. send(backtrack_CPA, backTo)

forward_check:
1. **if**(msg.SC > AgentView.SC)
2. **if**(**not** AgentView.consistent)
3. **if**(**not** contains(FC_CPA, AgentView))
4. AgentView.consistent ← true
5. **if**(AgentView.consistent)
6. **if (not**(update_AgentView(FC_CPA)))
7. send(Not_OK, unassigned_agents(AgentView))

process_Not_OK:
1. **if**(contains(AgentView, Not_OK))
2. AgentView ← Not_OK
3. AgentView.consistent ← false
4. **else if**(not-contains(Not_OK,AgentView))
5. **if**(msg.SC > AgentView.SC)
6. AgentView ← Not_OK
7. AgentView.consistent ← false

update_AgentView(*partial_assignment*):
1. adjust_AgentView(partial_assignment)
2. **if**(empty_domain)
3. AgentView ← *shortest_inconsistent_partial_assignment*
4. **return** *false*
5. **return** *true*

Fig. 2 AFC algorithm: backtracking and forward-checking processing

The next two functions in Fig. 2 implement the asynchronous forward-checking mechanism. Two types of messages can be received by an agent, FC_CPA and Not_OK (lines 9, 10 of the main function in Fig. 1).

Function ***forward_check*** is called when an agent receives a FC_CPA message. Since a FC_CPA message is relevant only if the message is an update of partial assignments received in previous messages, the SC value is checked to test the message relevance (line 1). "Older" SCs represent partial assignments that have already been checked within the partial assignment of the current (larger) SC of the receiving agent. When the $AgentView$ is inconsistent, the agent checks if its $AgentView$ is still relevant. If not, the $AgentView$ becomes consistent (lines 2–4). In case of a consistent $AgentView$, the agent updates its $AgentView$ and current-domains by calling the function $update\_AgentView$. If this causes an empty domain, the agent sends Not_OK messages to all agents which are unassigned in the inconsistent partial assignment found and stored in the AgentView (lines 6–7).

Function ***process_Not_OK*** checks the relevance of the received inconsistent partial assignment, with the $AgentView$. If the Not_OK message is relevant, it replaces the $AgentView$ by the content of the Not_OK message(lines 2–3).

Function ***update_AgentView***($partial\_assignment$) is called in case a CPA moving forward is received or a relevant FC_CPA. it sets the $AgentView$ and current domains to be consistent with the received partial assignment. In case of an empty domain, $update\_AgentView$ returns false and sets the $AgentView$ to hold the shortest inconsistent partial assignment.

Function ***adjust_AgentView***($partial\_assignment$) changes the content of the $AgentView$ to that of the received partial assignment. It also updates the current domains of the variables to be consistent with the $AgentView's$ new content.

The protocol of the AFC algorithm is designed so that *only one backtrack operation* is triggered by any number of Not_OK messages. This can be seen from the pseudo-code of the algorithm, in Figs. 1, 2 as follows:

- If a single agent discovers an empty domain, all Not_OK messages carry the same inconsistent partial assignment (Nogood) and each agent that receives such a Not_OK message has a consistent $AgentView$. In this case the CPA will finally reach an agent that holds an inconsistent $AgentView$, which is a subset of the set of assignments on the CPA. This CPA, at that step, will be sent back as a backtrack message.
- If two agents discover an empty domain as a result of receiving an identical FC_CPA and create Not_OK messages with identical inconsistent partial assignments. Other agents will receive two copies of the same Not_OK message. The second Not_OK message will be ignored since the Nogood it carries is the same as the one the receiving agent already holds. The rest of the processing will be the same as in the single empty domain case above.
- The general case is when two different agents send Not_OK messages that include two different inconsistent partial assignments. If one message is included in the other (i.e. a shorter Nogood), then the order of their arrival is irrelevant. If the shortest one arrives first, the long one is ignored. If the longer one arrives first the shorter one will replace it. If the two Not_OK messages include a different assignment to a common agent, then the receiving agent uses the SC on the messages to determine the more recent one and ignores the other.

At least one of the agents, that must receive and process the CPA, holds the Nogood (the creator of the nogood itself). This ensures that the backtrack operation will take place.

4 Correctness of AFC

A central fact that can be established immediately is that agents send forward only consistent partial assignments. This fact can be seen in lines 1, 2 and 8 of procedure *assign_CPA*. This implies that agents process, in procedures *receive_CPA* and *assign_CPA*, only consistent CPAs. Since the processing of CPAs in these procedures is the only means for extending partial assignments, the following lemma holds:

Lemma 1 AFC *extends only consistent partial assignments. The partial assignments are received via a* CPA *and are extended and sent forward by the receiving agent.*

The correctness of AFC includes soundness and completeness. The soundness of AFC follows immediately from the above Lemma. The only lines of the algorithm that report a solution are lines 3, 4 of procedure *assign_CPA*. Solution is reported when a CPA includes a complete and consistent assignment.

In order to prove the completeness and termination of AFC, one needs to make a few changes to function *assign_CPA*, in order to avoid stopping after finding the first solution. Assume therefore that instead of stopping after the first solution is found (line 5 of *assign_CPA*) the agent simply records the solution, removes its assignment and recalls function *assign_CPA*. The second needed change is to make the procedure of assigning values to variables concrete. This enables to prove the exhaustiveness of the assignments produced by AFC and to show termination. Assume that the function *add_local_assignments*, in line 1 of *assign_CPA* scans all values of a variable in some predefined order, until it finds a consistent assignment for the agent's variable. For the rest of the completeness proof it is assumed with no loss of generality that each agent holds exactly one variable.

Backtrack steps of AFC remove a single value from the domain of the agent that receives the backtrack message. This is easy to see in lines 8–10 of function *receive_CPA* in Fig. 1. The only way that a value removed by a backtrack step from agent A_i can be reassigned is after the CPA is sent further back to some agent A_j ($j < i$) and returns. Since there are a finite number of values in all agents domains, the following lemma is established.

Lemma 2 AFC *performs a finite number of backtrack steps.*

The termination of AFC follows immediately. Any infinite loop of steps of AFC must include an infinite number of backtrack steps and this contradicts Lemma 2.

AFC can in principle avoid sending forward consistent partial assignments through the mechanism of Not_OK messages. An agent that fails to find a value that is consistent with a received FC_CPA message sends a Not_OK message. This message may stop a recipient from trying to extend a valid and consistent assignment on a CPA. However, every Not_OK message is generated by a failure of the

function *update_AgentView* (lines 6, 7 of function *forward_check* in Fig. 2). The failure corresponds to a CPA that has no consistent value in the agent that generates the Not_OK message. Thus, the rejected CPA (i.e. its partial assignment) cannot be part of a solution of the $DisCSP$. This observation is stated by the next lemma.

Lemma 3 *Consistent CPAs that are not sent forward for extension because of a Not_OK message, cannot be extended to a solution (i.e. they are Nogoods).*

If AFC can be shown to process every consistent partial assignment (for a given order of agents/variables), this would establish the completeness of the algorithm. Completeness follows from this fact in analogy to the completeness proof for centralized backtracking in [7]. By Lemma 3, it is enough to prove completeness for the case where there are no Not_OK messages.

Assume by contradiction that there is a solution $S = (< A_1, V_1 >, < A_2, V_2 > \cdots < A_n, V_n >)$ that is not found by AFC. This means that some partial assignment of S is not sent forward by some agent. Let the longest partial assignment of S that is not sent forward be $S' = (< A_1, V_1 >, < A_2, V_2 > \cdots < A_k, V_k >)$ where $k < n$. S' is consistent, being a subset of S. There is at least one such partial assignment $(< A_1, V_1 >)$, performed by the first agent, because of its exhaustive scan of values. But, by lines 2, 8, 9 of function *assign_CPA*, agent A_k sends the partial assignment S' to the next agent because it is consistent. This contradicts the assumption of maximality of S'. This completes the correctness proof of algorithm AFC, soundness, termination and completeness.

5 Dynamic Ordering Heuristics

In centralized $CSPs$, dynamic variable ordering is known to be an effective heuristic for gaining efficiency [5]. A recent study has shown that the same is true for algorithms which perform synchronous (sequential) search on Distributed $CSPs$ [4]. Since the assignments in the AFC algorithm are performed sequentially by agents, as in the different versions of *Synchronous Backtracking* after each successful assignment an agent can choose a different agent to send the CPA to. The asynchronous forward-checking mechanism enables heuristics which are not possible in simple synchronous algorithms.

The different ordering heuristics can be divided into two groups, heuristics which can be performed without additional overhead in messages and heuristics that need this overhead.

5.1 Heuristics with No Additional Messages

The heuristics which do not need additional messages are either heuristics which can be performed in any synchronous backtrack algorithm or heuristics for which the additional information needed can be carried by the messages which are sent as part of the AFC algorithm. The following examples all fall into one of these characteristics:

– Random: an agent which successfully assigned its variables on the CPA chooses the next agent to send the CPA to randomly among all unassigned agents.

- Estimation of minimum domain size: Brito and Meseguer propose a heuristic for synchronous backtracking [4]. It is assumed that agents hold all the constraints they are involved in and know the initial size of the domains of other agents. In order to choose the next agent to send the CPA, the agents maintain two bounds for the size of the domain of each unassigned agent. Each agent that performs an assignment updates these bounds according to the number of conflicts its new assignment has with each of the unassigned agents. The lower bound of agent A_j is calculated as by the following formula:
  $l\_bound_j \leftarrow max(l\_bound_J, conflicts\_num(< A\_i, v_i >, A_j))$
  In words: the maximum between the former lower bound and the number of conflicts the new assignment has. The upper bound of agent A_j is calculated as follow:
  $u\_bound_j \leftarrow min(|D_j|, u\_bound_j + conflicts\_num(< A\_i, v_i >, A_j))$
  In words: the minimum between the size of the initial domain and the sum of the former upper bound and the number of conflicts.
  After all bounds are updated, if there exist an unassigned agent whose lower bound is the size of its domain or is higher than any other upper bound of any unassigned agent, the CPA is sent to it. Otherwise, it is sent to the agent with the highest upper bound among all unassigned agents.
- Nogood triggered: a heuristic inspired by *Dynamic Backtracking* [6]. The idea is to move forward the agent which initialized the backtrack operation. In AFC, in order to implement this idea an agent which receives a Not_OK message stores the ID of the agent it was received from. When the CPA is sent backwards the sending agent records the ID of the sender of the Not_OK message which triggered this backtrack operation on the CPA. The agent that receives the backtrack message, after replacing its assignment, sends the CPA to the triggering agent.

5.2 Heuristics with Additional Network Overhead

The following heuristics require additional messages which are not sent by the standard AFC algorithm:

- Actual current domain size: in their presentation of the estimated domain size heuristic, Brito and Meseguer report that it was found worth while to perform sequential backtrack instead of backjumping directly in order to enable agents to record their actual current domain size on the CPA [4]. In AFC this can be achieved without delaying the CPA. Each agent that filters its domain after receiving an FC_CPA sends messages which include its current domain size to each of the unassigned agents. Agents record the latest domain size they received from each agent and choose the smallest as the next agent to send the CPA to.
- A heuristic for variable and value ordering was presented in [13] for the direct backtracking version of AFC. Each agent holds a counter for each of the values in its domain and for each of the other agents. The counters are incremented as a result of a backtrack operation. When an agent encounters an empty domain it decreases the counter of the culprit agent it backtracks to. The agent that receives the backtrack message increments the counter of the sending agent. The sender of the backtrack message also checks for each value removed from its domain, which agent's assignment was the first to conflict with it. The counter

of each of these agents is incremented and a message is sent to them which indicates a possible conflict between the sending agent and the current value of the receiving agent. An agent that receives such a possible conflict message increases the counter of the sending agent and the counter of its current value. When agents assign their variables they choose the value with the lower counter in their domain. When an agent successfully assigns its variable, it chooses the agent with the highest counter among the unassigned agents to send the CPA to.

6 Improved Backtrack Method for AFC

In [13], an elegant method for initializing the backtrack operation in AFC was proposed. Instead of sending Not_OK messages to all unassigned agents in the inconsistent partial assignment, the agent whose domain emptied and triggers a backtrack operation, sends a *Backtrack* message to the last agent assigned in the inconsistent partial assignment (*Nogood*). All other agents receive a *Nogood* message which indicate that the former CPA is inconsistent. The receiver of the *Nogood* generates a new CPA and continues the search. The old CPA is detected as obsolete and discarded using the following method for time-stamping CPAs:

- The time-stamp is an array of counters, a single counter for each agent.
- An agent increments its counter when it performs an assignment.
- When two CPAs are compared, the more updated is the one whose time-stamp is larger lexicographically (i.e. the first different counter is larger).

Using this method agents which receive a Not_OK method that reveals the inconsistency of the former CPA and then receive the CPA itself simply terminate the old CPA. The only CPA which will not be terminated is the most updated according to the lexicographic time-stamp.

The improvement in performance of AFC with this method, is presented in Section 8.

7 Privacy in AFC

Privacy is the basic motivation for using a distributed algorithm for solving distributed problems [21]. However, the level of privacy required is not constant. Recent studies on privacy have shown that *DisCSPs* can be solved with complete privacy, i.e. no data loss by agents besides the result of the search [14, 16, 22]. However, this level of privacy requires the use of cryptographic and encryption tools which result in a substantial loss in efficiency. Thus different studies attempted to address this tradeoff between efficiency and privacy by trying to achieve lower levels of privacy using standard DisCSP methods [3, 25].

In the AFC algorithm as presented in this paper, agents reveal their assignments to their neighbors via the CPA and the FC_CPA messages. The assignments of agents are revealed to non-neighboring agents when they receive the CPA which includes assignments of all previous agents.

AFC can be adjusted to higher requirements of privacy if necessary. The most extreme way would be using the method presented in [14] which generates a fully secured protocol for any distributed algorithm in which agents hold a small state (AFC of course falls into this category). For lower requirements of privacy, such as not revealing assignments to non-neighboring agents, AFC can be adjusted in a number of ways. The simplest way would be to supply each pair of neighboring agents with an encryption key. Agents place their encrypted assignments on the CPA and their assignment is revealed only to their neighbors. If encryption methods cannot be used, agents in AFC can avoid placing their assignments on the CPA. Instead the assignments can be sent directly to the neighboring agents and the CPA will serve only as a token for the next assignment and for determining the order of the search. This adjustment however would require that in case of possible delays of messages, agents which receive the CPA wait for all the assignments of previous neighboring agents to arrive before they make an attempt to assign their variable and send the CPA forward.

8 Experimental Evaluation

To evaluate the performance of AFC, two sets of experiments were performed. The first investigates the difference between the variant versions of AFC, using different heuristics for ordering and performing different backtrack methods. In the second set of experiments, AFC is compared to one of the best performing $DisCSP$ algorithms, Asynchronous Backtracking (ABT) [2, 20].

8.1 Experimental Setup

All experiments use an asynchronous simulator in which agents are simulated by threads which communicate only through message passing. The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables. Such uniform random constraints networks of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 are commonly used in experimental evaluations of CSP algorithms (cf. [15, 18]) and $DisCSP$ algorithms (cf. [4, 24]).

The experimental setup included problems generated with 20 variables ($n = 20$) and 10 values ($k = 10$). The experiments include $DisCSPs$ with two different network density values $p_1 = 0.4$ and $p_1 = 0.7$. The value of p_2 was varied between 0.1 and 0.9, to cover all ranges of problem difficulty [15].

In order to evaluate the performance of distributed algorithms, two independent measures of performance are commonly used—run time in the form of steps of computation [9, 20] and communication load in the form of the total number of messages sent [9]. To take into account the local computation performed by agents in each step, computational cost can be evaluated in terms of non-concurrent constraints checks. The evaluation of the computational effort of distributed algorithms has to take concurrency into account. Non-concurrent constraint checks, in systems with no message delay, are counted by a method similar to that of Lamport [8, 11]. Every

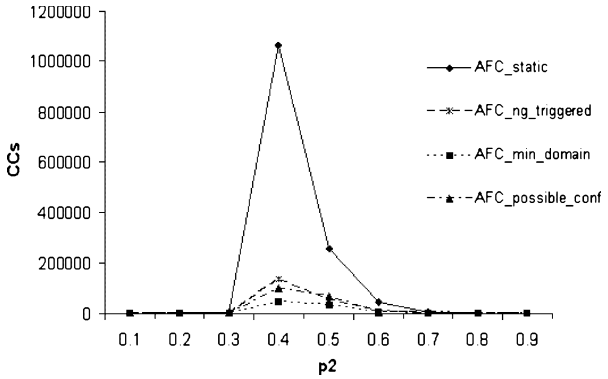

Fig. 3 Non-concurrent constraints with different ordering heuristics (p_1=0.4)

agent holds a counter of constraint checks. 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 concurrent search effort that is close to Lamport's logical time [8] is achieved.

The *NCCCs* measure is independent of the type or the implementation of the algorithms since it counts logic steps (*CCs*), it does not count logic steps which were performed concurrently, and it measures the cost of the algorithm step which would reveal the difference between different type of algorithms.

The total number of messages sent during the run of the algorithm is a common measure of network load for distributed algorithms [9].

8.2 Evaluation of Ordering Heuristics

Figure 3 presents the number of non-concurrent constraints checks performed by the AFC algorithm using different ordering heuristics on low density $DisCSPs$ ($p_1 = 0.4$). All three ordering heuristics improve the performance of the static order AFC. Figure 4 presents a closer look at the difference between the different heuristics, removing th static AFC. The best performing hezuristic is the minimal domain size

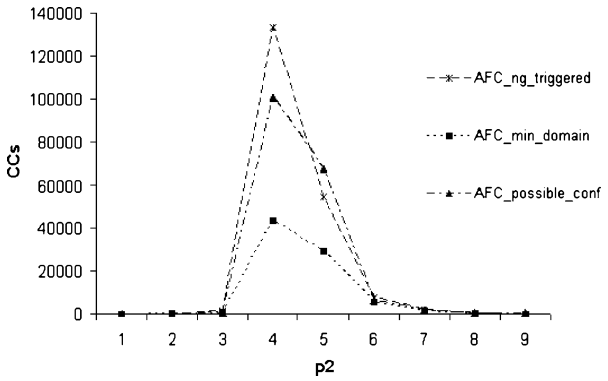

Fig. 4 Three ordering heuristics (p_1=0.4)

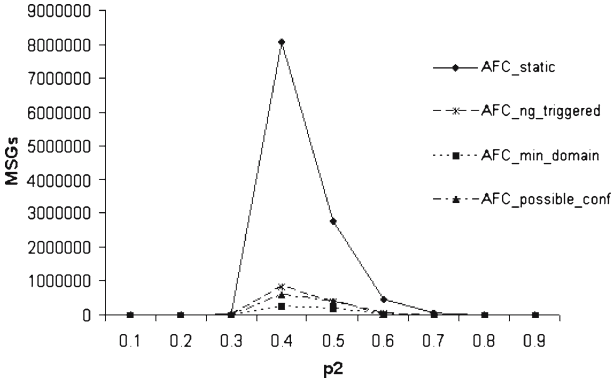

Fig. 5 Total number of messages sent by AFC with different ordering heuristics ($p_1 = 0.4$)

(*Min_Domain*) heuristic. *AFC* using the *Min_Domain* heuristic performs half the *NCCCs* of *AFC* with the possible conflict (*PC*) heuristic and a third of the *AFC* using the Nogood triggered (*NG*) heuristic. Figure 5 presents similar results for the measure of network load.

Figures 6 and 7 present the results of the same experiments, on *DisCSPs* with higher constraint density ($p_1 = 0.7$). Although the results are similar, the improvement gained by using ordering heuristics is less pronounced than for low density *DisCSPs*.

In order to evaluate the *AFC* algorithm with direct backjumping (*AFC_DBJ*) as presented in [13], it is compared to 'standard' *AFC*. Figures 8 and 9 compare *AFC* performing standard conflict based backjumping using the best ordering heuristic (*Min_Domain*) and *AFC* performing direct backjumping with the heuristic of [13]. *AFC* with direct backjumping performs better than standard *AFC*. The improvement more pronounced for higher density *DisCSPs* ($p_1 = 0.7$).

8.3 Comparison to Asynchronous Backtracking

The performance of Asynchronous Forward-checking (*AFC*) can be compared to Asynchronous Backtracking (*ABT*) Yokoo [20]. *ABT* is a complete asynchronous search algorithm, for which assignments are performed asynchronously. In the *ABT*

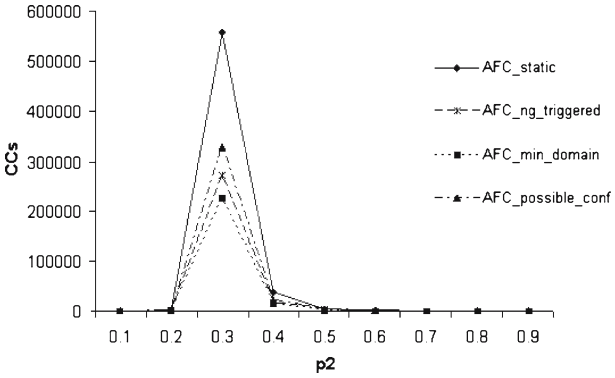

Fig. 6 Non-concurrent constraints checks for different ordering heuristics solving high density DisCSPs ($p_1 = 0.7$)

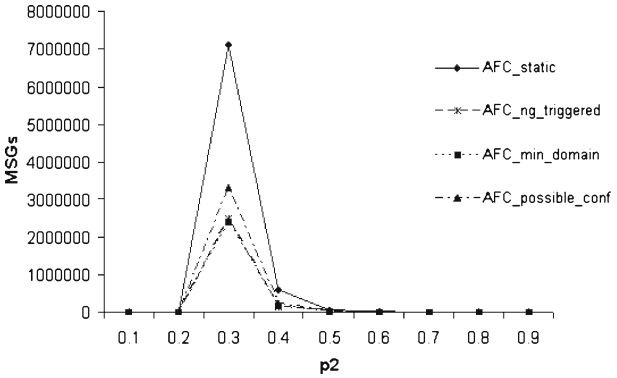

Fig. 7 Total number of messages for different ordering heuristics ($p_1 = 0.7$)

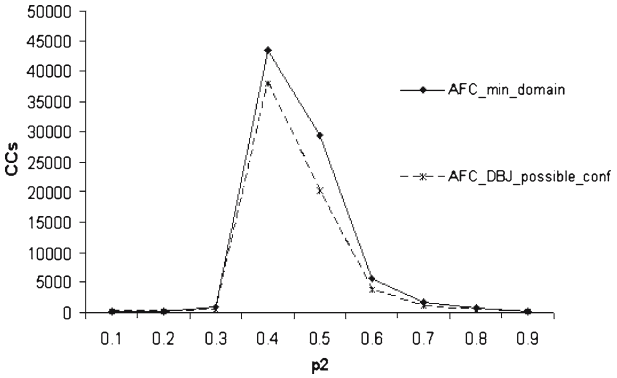

Fig. 8 Non-concurrent constraints checks performed by AFC using different backtracking methods solving low density DisCSPs (p1=0.4)

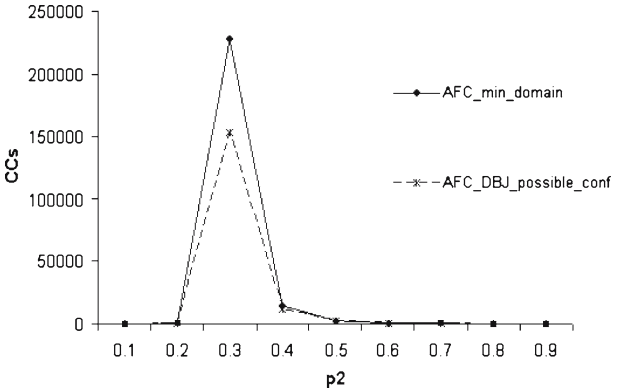

Fig. 9 Non-concurrent constraints checks performed by AFC using different backtracking methods solving high density DisCSPs (p1=0.7)

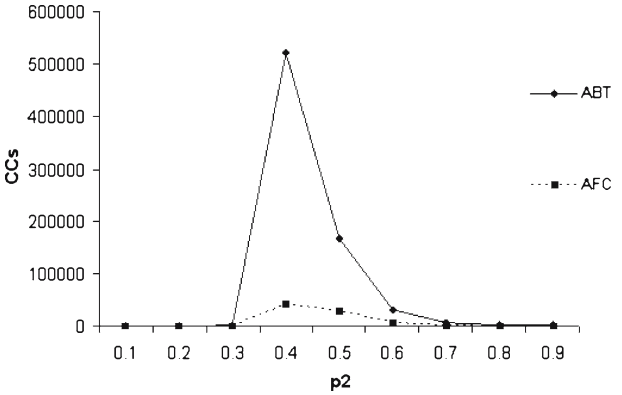

Fig. 10 Non-concurrent constraint checks performed by AFC and ABT solving low density *DisCSPs* ($p_1 = 0.4$)

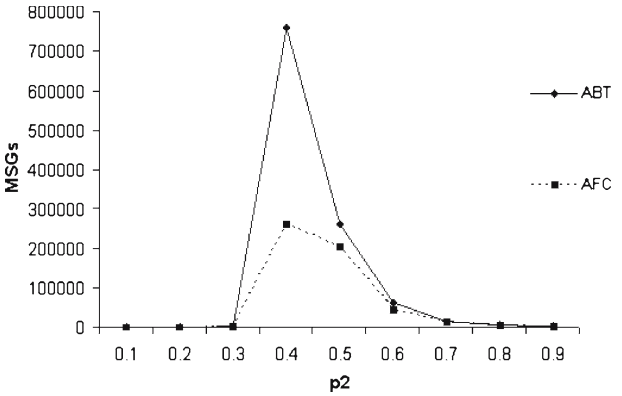

Fig. 11 Total number of messages sent by AFC and ABT solving low density *DisCSPs* ($p_1 = 0.4$)

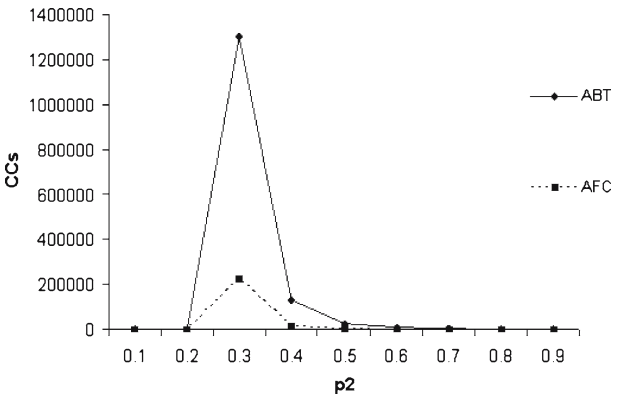

Fig. 12 Non-concurrent constraint checks performed by AFC and ABT ($p_1 = 0.7$)

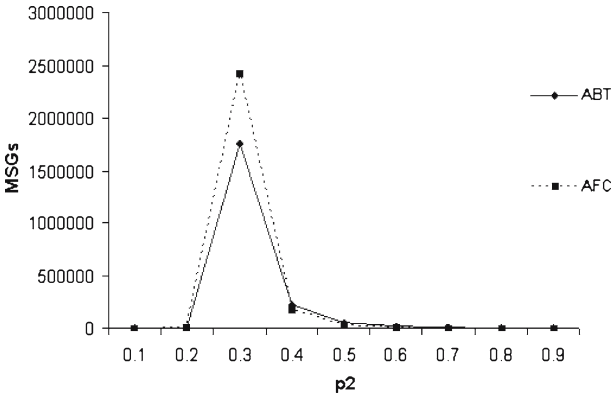

Fig. 13 Total number of messages sent by AFC and ABT ($p_1 = 0.7$)

algorithm agents assign their variables asynchronously, and send their assignments in *ok*? messages to other agents to check against constraints. A fixed priority order among agents is used to break conflicts. Agents inform higher priority agents of their inconsistent assignment by sending them the inconsistent partial assignment in a *Nogood* message. In our implementation of ABT, the *Nogoods* are resolved and stored according to the method of dynamic backtracking (DB), presented in [1, 6]. Based on Yokoo's suggestions [20] and the results of recent studies of ABT [2, 23], the agents read, in every step, all the messages in their mailbox before performing computation. However, since the simulator used is asynchronous in our experiments, the agents read in the beginning of every step the messages which their mail-box contain but can't wait for additional messages before they begin the computation phase. The performance of ABT is compared to AFC, which uses the best performing heuristic according to the experiments above, *Min_Domain*.

Figures 10 and 11 present the results in run-time and in network load of AFC and ABT for low density *DisCSPs*. AFC outperforms ABT by a factor of 10 in number of *NCCCs* and by a factor of 3 in the total number of messages. Figures 12 and 13 present the results of the same comparison for higher density *DisCSPs*. For higher density networks, the factor of improvement of AFC over ABT in run-time is only 7. In network-load the results are slightly in favor of ABT.

9 Discussion

In asynchronous backtrack algorithms agents attempt to speed up the search by assigning their variables concurrently. Recent studies have shown that versions of synchronous backtracking that perform backjumping according to conflict sets and order the assignments according to intelligent heuristics achieve equal or even better performance than that of asynchronous backtracking [4, 23]. In AFC agents are informed of the consistent partial assignments of other agents and asynchronously filter their domains accordingly. Domain filtering triggers early backtracking operations. In contrast, local data structures in asynchronous backtracking often holds an inconsistent partial assignment.

The use of a synchronous assignment procedure, enables an easy implementation of different ordering heuristics for AFC. The results in Figs. 3 to 7 present a strong

improvement in two independent measures of performance that is achieved by the use of these ordering heuristics.

The fact that the ratio of improvement of AFC over ABT grows with problem difficulty can be explained intuitively. Problem difficulty is known to be correlated with the number of solutions on random constraint networks [18]. Fewer solutions mean that a larger fraction of all partial assignments will fail. In asynchronous backtracking, each such "due to fail" assignment generates messages to multiple agents and triggers their further assignments and message passing. The reported experiments demonstrate that when there are fewer solutions it is more efficient to generate consistent partial assignments, as does the AFC algorithm.

10 Conclusions

A new distributed search algorithm on *DisCSP*s has been presented. The asynchronous forward-checking (AFC) algorithm keeps a unique partial assignment at all times and sends it to all agents to perform forward checking. A current partial assignment (CPA) is passed among all agents and is always consistent. Agents add their consistent assignments to the CPA, if such an assignment can be found. The concurrency of AFC springs from the fact that *forward-checking messages are processed concurrently*. In other words, copies of every valid CPA are sent forward, to unassigned agents, to perform forward-checking. When an inconsistency is discovered by an agent that is still not on the CPA (i.e. an unassigned agent), a Not_OK message is sent to all unassigned agents. The Not_OK messages trigger a single backtrack operation.

The main conclusion of the present study is that coordination of the assignments performed in distributed search enhances the efficiency of the search process. The performance of asynchronous forward-checking generates a more efficient search than asynchronous backtracking. One major advantage of $DisCSP$ algorithms that perform sequential assignments is the ability to use ordering heuristics. Three ordering heuristics were used in the evaluation of AFC. All three heuristics improve the performance of AFC. The best heuristic, *Min_Domain*, that is proposed here for the first time enables AFC to outperform ABT by almost an order of magnitude. It turns out that the advantages of dynamic variable ordering are enough to overcome the price of coordination that is needed for forward checking [10].

References

1. Bessiere, C., Maestre, A., & Messeguer, P. (2001). Distributed dynamic backtracking. In *Proc. Workshop on Distributed Constraint of IJCAI01*.
2. Bessiere, C., Maestre, A., Brito, I., & Meseguer, P. (January 2005). Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence, 161*(1–2), 7–24.
3. Brito, I., & Meseguer, P. (September 2003). Distributed forward checking. In *Proc. CP-2003*, (pp. 801–806). Ireland.
4. Brito, I., & Meseguer, P. (September 2004). Synchronous, asnchronous and hybrid algorithms for discsp. In *Workshop on Distributed Constraints Reasoning* (*DCR-04*) CP-2004, Toronto.
5. Dechter, R. (2003). *Constraints Processing*. Morgan Kaufmann.
6. Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research, 1*, 25–46.

7. Kondrak, G., & van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence, 21*, 365–387.
8. Lamport, L. (April 1978). Time, clocks, and the ordering of events in distributed system. *Communication of the ACM, 2*, 95–114.
9. Lynch, N. A. (1997). *Distributed Algorithms*. Morgan Kaufmann.
10. Meisels, A., & Razgon, I. (2002). Distributed forward-checking with conflict-based backjumping and dynamic ordering. In *Proc. CoSolv workshop, CP02*. Ithaca: NY.
11. Meisels, A., Razgon, I., Kaplansky, E., & Zivan, R. (July 2002). Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, (pp. 86–93). Bologna.
12. Meseguer, P., & Jimenez, M. A. (September 2000). Distributed forward checking. In *Proc. CP-2000 Workshop on Distributed Constraint Satisfaction*, Singapore.
13. Nguyen, T., Sam-Hroud, D., & Faltings, B. (September 2004). Dynamic distributed backjumping. In *Proc. 5th workshop on distributed constraints reasoning DCR-04*, Toronto.
14. Nissim, K., & Zivan, R. (2005). Secure discsp protocols—from centralized towards distributed solutions. In *Proc. 6th workshop on Distributed Constraints Reasoning, DCR-05*, Edinburgh.
15. Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence, 81*, 81–109.
16. Silaghi, M. C. (2002). *Asynchronously Solving Problems with Privacy Requirements*. PhD Thesis, Swiss Federal Institute of Technology (EPFL).
17. Silaghi, M. C., & Faltings, B. (January 2005). Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence, 191*(1–2), 25–54.
18. Smith, B. M. (1996). Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence, 81*, 155–181.
19. Solotorevsky, G., Gudes, E., & Meisels, A. (1996). Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, (pp. 561–562). New Hamphshire.
20. Yokoo, M. (2000). Algorithms for distributed constraint satisfaction problems: a review. *Autonomous Agents and Multi-Agent Sys., 3*, 198–212.
21. Yokoo, M. (2000). *Distributed Constraint Satisfaction Problems*. Springer Berlin Heidelberg New York.
22. Yokoo, M., Suzuki, K., & Hirayama, K. (January 2005). Secure distributed constraints satisfaction: Reaching agreement without revealing private information. *Artificial Intelligence, 161*(1–2), 229–246.
23. Zivan, R., & Meisels, A. (December 2003) Synchronous vs asynchronous search on discsps. In *Proc. 1st European Workshop on Multi Agent System, EUMAS*, Oxford.
24. Zivan, R., & Meisels, A. (2004). Concurrent dynamic backtracking for distributed csps. In *CP-2004*, (pp. 782–787). Toronto.
25. Zivan, R., & Meisels, A. (2005). Asynchronous backtracking for asymmetric discsps. In *Proc. 6th workshop on Distributed Constraints Reasoning, DCR-05*, Edinburgh.