Asynchronous Forward-Checking on DisCSPs

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

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. An experimental comparison of *AFC* to Asynchronous BackTracking (ABT), on randomly generated *DisCSP*s with different forms of communication is presented. *AFC* outperforms ABT on systems with a random message delay by a large factor on the harder instances of random *DisCSP*s. This result holds for both measures of performance, total run time in the form of computation steps and network load in the form of the total number of messages sent.

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 (cf. [Yokoo2000, Solotorevsky et. al.1996]). 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 [Yokoo2000, Bessiere et. al.2001].

Several asynchronous backtracking algorithms on $DisCSP$s have been proposed in recent years [Yokoo et. al.1998, Bessiere et. al.2001, Silaghi et. al.2001]. 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 [Yokoo2000, Bessiere et. al.2001]. 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 one consistent partial assignment and processes it 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)*. The *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 the *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 CPAs. This is in contrast to Distributed Forward Checking [Meseguer and Jimenez2000] which filters domains against *asynchronous assignments by other agents*. As in all asynchronous backtracking algorithms, a set of asynchronous assignments is not in general consistent. The synchronous method of performing assignments in AFC may generate some confusion with a simple distributed search algorithm on $DisCSP$s - *Synchronous Backtrack* (cf. [Yokoo2000]). 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 ABT (section 5).

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 algorithm is described in detail in section 3 and its correctness is proven in section 4. The performance of *AFC* is compared to that of asynchronous backtracking (ABT) on randomly generated *DisCSP*s with different message delays. *AFC* outperforms ABT on systems with random message delay by a large factor on hard instances of random problems. This is true for both measures of performance: the number of steps of computation and the number of messages sent (see section 5). A discussion of the differences of the *AFC* algorithm from asynchronous backtracking and of its improved performance is presented in section 6. Our conclusions, that increasing the coordination among agents of a distributed search algorithm improves its performance, are presented in section 7.

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 vari-

ables that belong to different agents [Yokoo et. al.1998, Solotorevsky et. al.1996]. In addition each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair $< var, val >$, where var is a variable of some agent and val is a value from var's domain that is assigned to it. A *compound label* is a set of assignments of values to a set of variables. A **solution** P to a *DisCSP* is a compound label that includes all variables of all agents, that satisfies all the constraints.

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 [Yokoo2000]. 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. [Yokoo et. al.1998, Bessiere et. al.2001]).

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.

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 consis-

tent) 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 figure 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 Figure 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 is valid with respect to the received CPA, this means that a backtrack of the *CPA* has to be performed. For an inconsistent $AgentView$ the agent checks the relevance of the partial assignment held in its $AgentView$ (line 3). In case the $AgentView$ contains a partial assignment of the received CPA this means that the received CPA is inconsistent and the agent initiates a backtrack. Otherwise, the $AgentView$ is not valid and is turned 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 $assign\_CPA$.

- ***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**(CPA = 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_sloution
  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

Figure 1: AFC Algorithm - receive and 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 10-11). 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 Figure 2. Function ***backtrack*** is called when the agent is holding the CPA in one of two cases. Eather 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.

The next two functions in Figure 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 Figure 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 unassigned agents (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 shortest inconsistent assignment (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 Figures 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

- ***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(FC_CPA))

- ***process_Not_OK***:
  1. **if**(contains(AgentView, Not_OK))
  2. AgentView ← $shortest\_inconsistent\_partial\_assignment$
  3. AgentView.consistent ← false
  4. **else**
  5. **if**(msg.SC $>$ AgentView.SC)
  6. AgentView ← $shortest\_inconsistent\_partial\_assignment$
  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** $true$

Figure 2: AFC Algorithm - backtracking and forward-checking processing

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 caries 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 7 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 of stoping 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 Figure 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, the following lemma is established.

Lemma 2. AFC *can perform 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 Figure 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, can not 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 [Kondrak and vanBeek1997]. 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 Experimental Evaluation

To evaluate the performance of *AFC*, it is compared to Asynchronous BackTracking (ABT) [Yokoo et. al.1998, **?**]. Since the ABT algorithm was only presented for solving $DisCSP$s with one variable per agent, in all our experiments, algorithms were run on randomly generated *DisCSP*s, where each agent holds one variable.

5.1 Experimental Setup

In all our experiments we use a simulator in which agents are simulated by threads which communicate only through message passing. The network of constraints, in each of our 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. [Prosser1996, Smith1996]).

Our setup included problems generated with 10 variables ($n = 10$) and 10 values ($k = 10$). The value of network density p_1 in all our experiments was set to 0.7. The value of p_2 was varied between 0.1 and 0.9, to cover all ranges of problem difficulty [Prosser1996].

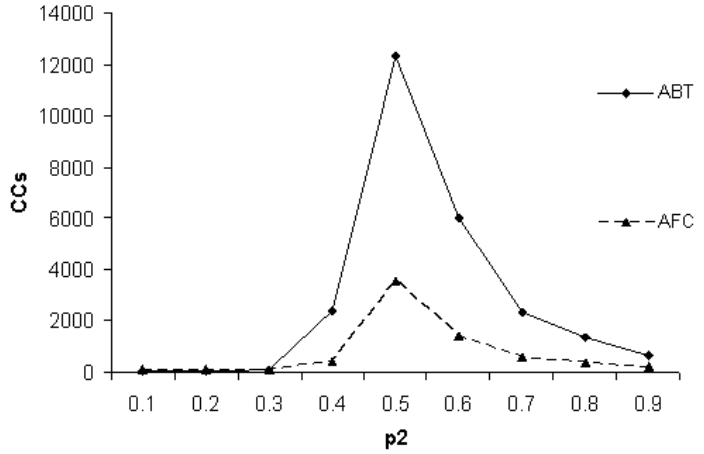

Figure 3: Number of concurent constraints checks in AFC, and ABT

In order to evaluate the performance of the distributed algorithms, two independent measures of performance are commonly used - total run time in the form of computation steps [Lynch1997, Yokoo2000] and communication load in the form of the total number of messages sent [Lynch1997]. In order to take into account the local computation performed by agents in each step, computational cost can be evaluated in terms of concurrent constraints checks. The evaluation of the computational effort of distributed algorithms has to take concurrency into account. Concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [1, Meisels et. al.2002]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent's counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search, a concurrent search effort that is close to Lamports logical time [1] is achieved. The same method can be used for counting concurrent constraints checks.

On systems with message delays, the situation is different. In the presence of concurrent computation, the time of message delays must be added to the total algorithm time *only if no computation was performed concurrently*. To achieve this goal, the simulator counts message delays in terms of concurrent constraints checks and adds them to the accumulated count only when no computation is performed concurrently [Zivan and Meisels2004a].

The total number of messages sent during the run of the algorithm is a common measure of performances of distributed algorithms [Lynch1997, Attiya and Welch1998].

5.2 Comparison to Asynchronous backtracking

The performance of AFC is compared to Asynchronous BackTracking (ABT) [Yokoo et. al.1998]. In the ABT 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 presented in [Bessiere et. al.2001]. Based on Yokoo's suggestions [Yokoo2000] the agents read, in every step, all messages in their mailbox before performing computation.

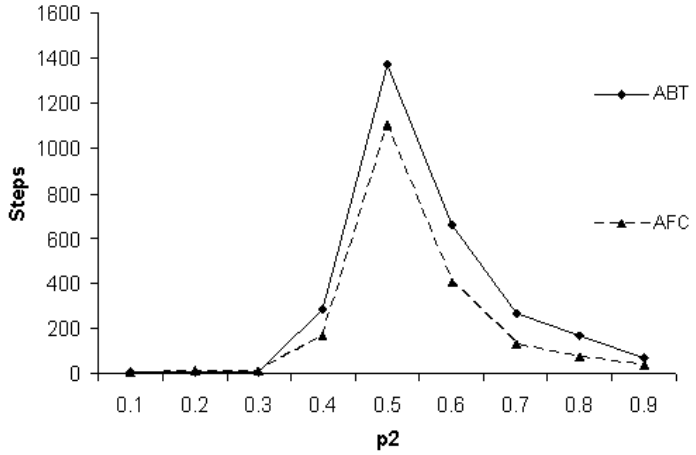

Figure 4: Number of concurent steps in AFC, and ABT

Although it is clear that a version of ABT that reads all messages received in every step of computation can be much more efficient, such a property makes the efficiency of ABT dependent on the form of communication. The consistency of the $Agent\_view$ held by an agent, with the actual state of the system before it begins the assignment attempt is affected directly by the number and the relevance of the messages it received up to this step. When the message delay is random, as in a real world scenario, the performance of ABT deteriorates [Fernandez et. al.2002].

In our experiments we compare AFC to ABT on systems with no message delay and on systems with random message delays of between 5 and 15 logical constraints checks.

5.3 Experimental Results

Figures 3, 4 present a comparison of the computational effort performed by AFC and ABT on randomly generated $DisCSPs$. The advantage in concurrent constraints checks (figure 3) of AFC over ABT is more pronounced then in concurrent steps (figure 4). This indicates that the distributed procedure which maintains local consistency in AFC is efficient. It needs fewer constraints checks per computation step on the average.

Figure 5 shows that the message load of AFC is smaller than of ABT.

In the second set of experiments each message was delayed by a random number of logical concurrent constraints checks, between 5 and 15. The results in figures 6, 7 show as expected, that the performance of both algorithms deteriorates with message delays. The difference in concurrent constraints checks, between AFC and ABT is smaller on systems with random message delay. The difference in the total number of messages however, is larger (figure 7). In both cases AFC has an advantage over ABT.

6 Discussion

In asynchronous backtrack algorithms agents attempt to speed up the search by assigning their variables concurrently. Since agents do not wait for the consistent assignments of other agents the local data structure in asynchronous backtracking often holds an inconsistent assignment.

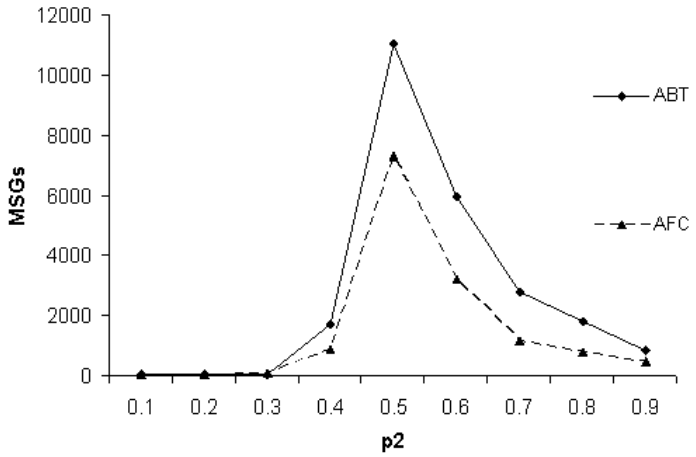

Figure 5: Total Number of messages sent by AFC, and ABT

In the AFC algorithm agents perform local computation concurrently against consistent partial assignment. When counting constraints checks, AFC outperforms asynchronous backtracking by a large factor. This can be explained by the fact that the amount of computation agents need to perform per each assignment is much smaller in AFC. When messages are delayed ABT cannot read multiple messages before performing computation. Therefore the actual steps it performs and the number of messages it sends grow by a significant factor. AFC is slowed down by message delay when the CPA is delayed, or a Not_OK message indicating a need for backtrack is delayed, while the CPA is moving forward. In other words, message delay disables some of the pruning of AFC. It delays the backtrack trigered by forward checking.

These are interesting results in view of the fact that AFC performs only assignments that are consistent with a unique partial assignment. It evolves the current partial assignment in a series of steps that are never performed concurrently. The partial assignment (on the CPA) traverses a chronological backtrack search tree [Kondrak and vanBeek1997] and the only concurrency of the AFC algorithm arises from the processing of forward-checking messages and reactions to their failure. It is interesting that the concurrency of performing forward checking is enough to outperform asynchronous backtracking.

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 [Smith1996]. Fewer solutions mean that a larger fraction of proposed 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 just one assignment, as does the AFC algorithm.

Asynchronous backtracking performs better when $AgentViews$ are close to the real state of the system. Reading all messages before computing [Yokoo2000] in systems with no message delays generate $AgentViews$ that correspond more accurately to the state of the system. But, this represents the fact that all messages arrive instantaneously. Systems with message delay simulate the real world where not all messages arrive at the same time. The performance of ABT and AFC deteriorate due to message delay. Agents in ABT perform more steps of computation and send more messages while the actions of agents in AFC are delayed.

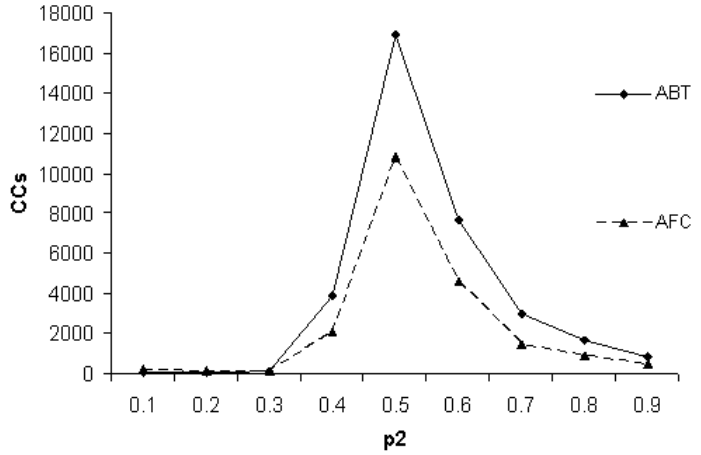

Figure 6: Number of logical concurent constraints checks in AFC, and ABT running on systems with random message delay

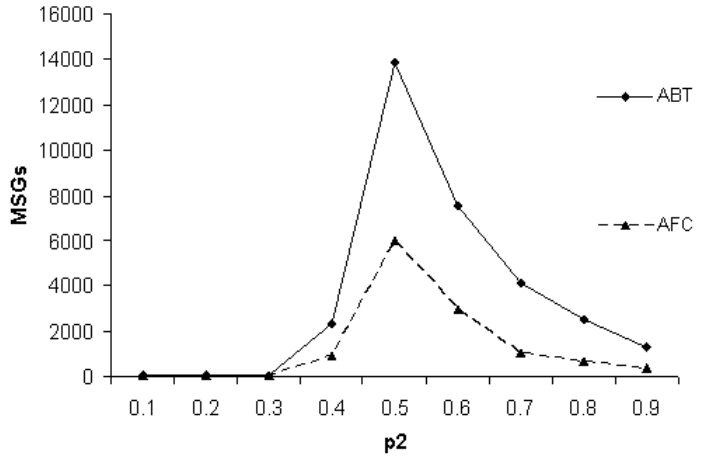

Figure 7: Total Number of messages sent by AFC, and ABT running on systems with random message delay

7 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. The 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. Otherwise, agents send the CPA back to an assigned agent, to revise its assignment on the CPA. 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 the assignments performed in distributed search enchances the efficiency of the search process. The performance of asynchronous forward-checking generates a much more efficient search than asynchronous

backtracking. The accumulated evidence for this phenomenan is massive. It occurs for three different measures of performance - steps of computation, concurrent constraints checks (CCCs), and network load. The clear evidence for the efficient behavior of AFC, compared to asynchronous backtracking, adds to former investigations on randomly generated $DisCSP$s and especially on harder problem instances. Similar behaviour was found for a distributed search algorithm that performs forward checking and orders variables dynamically [Meisels and Razgon2001]. It turns out that the advantages of dynamic variable ordering are enough to overcome the price of coordination that is needed for forward checking [Meisels and Razgon2001]. In a different study a concurrent backtrack algorithm on $DisCSP$s searches concurrently in several non intersecting search spaces, but performs in each search space a simple chronological backtrack search [Zivan and Meisels2002]. Recent experiments show that concurrent backtrack performs fewer steps of computation and sends fewer messages than asynchronous backtracking on random $DisCSP$s[Zivan and Meisels2004].

References

[Attiya and Welch1998] H. Attiya and J. Welch. Distributed Computing. $McGraw - Hill$, 1998.

[Bessiere et. al.2001] C. Bessiere, A. Maestre and P. Messeguer. Distributed Dynamic Backtracking. *Proc. Workshop on Distributed Constraints, IJCAI-01*, Seattle, 2001.

[Fernandez et. al.2002] C. Fernandez, R. Bejar, B. Krishnamachari, K. Gomes Communication and Computation in Distributed CSP Algorithms. *Proc. Principles and Practice of Constraint Programming, CP-2002*, pp. 664-679, Ithaca NY USA, July, 2002.

[Kondrak and vanBeek1997] G. Kondrak and P. vanBeek. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, vol. 89, pp.365-87, 1997.

[1] L. Lamport Time, clocks and the ordering of events in a distributed system. *Comm. of ACM*, vol. 21, pp.558-565, 1978.

[Lynch1997] N. A. Lynch. Distributed Algorithms. *Morgan Kaufmann Series*, 1997.

[Meisels and Razgon2001] A. Meisels and I. Razgon. Distributed Forward-checking with Dynamic Ordering, *Proc. DCR Workshop IJCAI-01*, Seattle, July, 2001.

[Meisels et. al.2002] A. Meisels et. al. Comparing performance of Distributed Constraints Processing Algorithms. *Proc. DCR Workshop, AAMAS-2002* , pp. 86-93, Bologna, July, 2002.

[Meseguer and Jimenez2000] P. Meseguer and M. A. Jimenez. Distributed Forward Checking. *Proc. DCR Workshop, CP-2000*, Singapore, September, 2000.

[Prosser1993] P. Prosser. Hybrid Algorithm for the Constraint Satisfaction Problem, *Computational Intelligence*, vol. 9, pp. 268-299, 1993.

[Prosser1996] P. Prosser An empirical study of phase transition in binary constraint satisfaction problems *Artificial Intelegence*, vol. 81, pp. 81-109, 1996.

[Silaghi et. al.2001] M. C. Silaghi, D. Sam-Haroud and B. Faltings. Asynchronous Consistency Maintenance. In *2nd Asia-Pacific IAT*, pp. 100-104, Maebashi, Japan, 2001.

[Smith1996] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. In *Artificial Intelegence*, vol. 81, pp. 155-181, 1996.

[Solotorevsky et. al.1996] G. Solotorevsky, E. Gudes and A. Meisels. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). *Constraint Processing-96*, New Hamphshire, October 1996.

[Yokoo et. al.1998] M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara. Distributed Constraint Satisfaction Probelm: Formalization and Algorithms. *IEEE Trans. on Data and Kn. Eng.*, vol. 10(5), pp. 673-685, 1998.

[Yokoo2000] M. Yokoo. Algorithms for Distributed Constraint Satisfaction: A Review. *Autons Agents Multi-Agent Sys 2000*, vol. 3(2), pp. 198-212, 2000.

[Zivan and Meisels2002] R. Zivan and A. Meisels. Parallel Backtrack Search on DisCSPs. *Proc. DCR Workshop, AAMAS-2002* , pp. 202-208, Bologna, July, 2002.

[Zivan and Meisels2004] R. Zivan and A. Meisels. Concurrent Backtrack Search on DisCSPs. *to apear in Flairs-2004 available at:* http://www.cs.bgu.ac.il/˜zivanr

[Zivan and Meisels2004a] R. Zivan and A. Meisels. Message delay and DisCSP search algorithms. *submit to Flairs-2003*