Message delay and Asynchronous DisCSP search[*]

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

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

Abstract. Distributed constraint satisfaction problems (DisCSPs) are composed of agents, each holding its own variables, that are connected by constraints to variables of other agents. Due to the distributed nature of the problem, message delay can have unexpected effects on the behavior of distributed search algorithms on DisCSPs. This has been shown in experimental studies of asynchronous backtracking algorithms [1, 9].
To evaluate the impact of message delay on the run of DisCSP search algorithms, a model for distributed performance measures is presented. The model counts the *number of non concurrent constraints checks*, to arrive at a solution, as a non concurrent measure of distributed computation. A simpler version measures distributed computation cost by the number of non-concurrent steps of computation. An algorithm for computing these distributed measures of computational effort is described. The realization of the model for measuring performance of distributed search algorithms is a simulator which includes the cost of message delays.
The performance of two asynchronous search algorithms is measured on randomly generated instances of DisCSPs with delayed messages. The Asynchronous Weak Commitment (AWC) algorithm and Asynchronous Backtracking (ABT). The intrinsic reordering process of AWC dictates a need for a more complex count of non-concurrent steps of computation. The improved counting algorithm is also needed for Dynamic ordered ABT. The delay of messages is found to have a strong negative effect on AWC and a smaller effect on dynamically ordered ABT.

Key words: Distributed Constraint Satisfaction, Search, Distributed AI.

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. [13, 11]). Agents check the value assignments to their variables for local consistency and

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

exchange messages among them, to check consistency of their proposed assignments against constraints with variables that belong to different agents [13, 2].

Search algorithms on DisCSPs are run concurrently by all agents and their performance must be measured in terms of distributed computation. Two measures are commonly used to evaluate distributed algorithms - run time, which measures the computational effort and network load [5]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [13]). In order to take into account the effort an agent makes during its local assignment the computational effort can be measured by the number of constraints checks that agents perform. However, care must be taken to measure the *non-concurrent* constraints checks. In other words, count computational effort of concurrently running agents *only once* during each concurrent running instance ([6, 8]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [5].

The first attempts to compare run times of distributed search algorithms on DisCSPs used a synchronous simulator and instantaneous message arrival. During one step of computation (cycle) of the simulator all messages of all agents are delivered and all resulting computations by the receiving agents are completed [13]. The number of these synchronous steps of computation in a standard simulator served to measure the non-concurrent run-time of a DisCSP algorithm [13]. It is clear that the comparison of asynchronous search algorithms by synchronizing them to run on a simulator is not satisfactory. In fact, comparing concurrent run-times of distributed computations must involve some type of asynchronous time considerations [4, 6].

The need to define a non-concurrent measure of time performance arises even for an optimal communication network, in which messages arrive with no delay. It turns out that for ideal communication networks one can use the number of non-concurrent constraints checks (NCCCs), for an implementation independent measure of non-concurrent run time [6]. When messages are not instantaneous, the problem of measuring distributed performance becomes more complex. On realistic networks, in which there are variant message delays, the time of run cannot be measured in steps of computation. Take for example Synchronous Backtracking (SBT) [13]. Agents in SBT perform their assignments one after the other, in a fixed order, simulating a simple backtrack algorithm. Since all agents are completely synchronized and no two agents compute concurrently, the number of computational steps is not affected by message delays. However, the effect on the run time of the algorithm is completely cumulative (delaying each and every step) and is thus large (see section 6 for details).

The present paper proposes a general method for measuring run time of distributed search algorithms on DisCSPs. The method is based on standard methods of asynchronous measures of clock rates in distributed computation [4] and uses constraints checks as a logical time unit [6]. In order to evaluate the impact of message delays on DisCSP search algorithms, we present an *Asynchronous Message Delay Simulator* ($AMDS$) which measures the logical time of the algorithm run. The $AMDS$ measures run time in non-concurrent steps of computation or in non-concurrent constraints checks and simulates message delays accordingly. The $AMDS$ and its underlying asynchronous measuring algorithm for comparing concurrent running times is described in detail in section 3. The validity of the $AMDS$' counting algorithm, to measure concurrent logical time, is proved in section 4. It can simulate systems with different types of message delays. From fixed message delays, through random message delays, to systems in which the length of the delay of each message is dependent on the current load of the network. The delay is measured in non-concurrent computation steps (or non-concurrent constraints checks). The final logical time that is reported as the cost of the algorithm run, includes steps of computation which were actually performed by some agent, and computational steps which were added as message delay simulation while no computation step was performed concurrently (see section 3).

The $AMDS$ presented in section 3 enables a deeper exploration of the behavior of different search algorithms for DisCSPs on systems with different message delays. Message delays emphasize properties of algorithms which are not apparent when the algorithms are run in a system with perfect communication. Experimental evidence for such behavior was found recently for asynchronous backtracking algorithms [1, 9]. The study of [1] measured run times on a multi-machine implementation of the compared algorithms. While serving as a first attempt to study the impact of communication delays on DisCSP algorithms, such an implementation does not enable simple duplication of experiments, for diverse algorithms and measures, as does the present well-defined simulation algorithm.

Measuring asynchronous backtracking search algorithms with dynamic agent ordering [12, 17] generates an additional problem which is not present for standard $DisCSP$ algorithms. For both the AWC algorithm of [12] and the *Dynamic Ordering ABT* algorithm of [17], assignment messages are sent by agents to all their neighbors including higher priority neighbors. Such messages carry information which does not trigger and is not evaluated in the following computation of the receiving agent. The method of the $AMDS$ simulator proposed in this study ensures that logical steps (constraints checks or computation steps) are counted twice only if they could not have been performed concurrently.

The plan of the paper is as follows. Distributed constraint satisfaction problems ($DisCSPs$) are presented in section 2. A detailed introduction of the simulator that is used in our experiments, and of the method of evaluating the run time of $DisCSP$ algorithms in the presence of message delays, is presented in section 3. Section 4 contains a proof of the validity of the simulating algorithm. Section 5 presents the two $DisCSP$ search algorithms Asynchronous Backtracking (ABT) and Asynchronous Weak Commitment search (AWC). Section 6 presents extensive experimental results, comparing the two algorithms on randomly generated $DisCSP$s with different types of message delays. A discussion of the performance and advantages of the algorithms, on different $DisCSP$ instances and communication networks, is presented in section 7. Our conclusions are in section 8.

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 [3]. A **binary constraint** R_{ij} between any two variables X_j and X_i is defined as: $R_{ij} \subseteq D_j \times D_i$. In a distributed constraint satisfaction problem *DisCSP*, the agents are connected by constraints between variables that belong to different agents (cf. [13, 11]). 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 *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is an assignment that includes all variables of all agents, that is consistent with all 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 [13].

The delay in delivering a message is assumed to be finite [13]. One simple protocol for checking constraints, that appears 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. The following assumptions are routinely made in studies of $DisCSP$s and are assumed to hold in the present study [13, 2].

1. All agents hold exactly one variable.
2. The amount of time that passes between the sending of a message to its reception is finite.
3. Messages sent by agent A_i to agent A_j are received by A_j in the order they were sent.
4. Every agent can access the constraints in which it is involved and check consistency against assignments of other agents.

3 Simulating search on DisCSPs

The standard model of Distributed Constraints Satisfaction Problems has agents that are autonomous asynchronous entities. The actions of agents are triggered by messages that are passed among them. In real world systems, messages do not arrive instantaneously but are delayed due to networks properties. Delays can vary from network to network (or with time, in a single network) due to networks topologies, different hardware and different protocols used. To simulate asynchronous agents, the simulator implements agents as *Java Threads*. Threads (agents) run asynchronously, exchanging messages by using a common mailer. After the algorithm is initiated, agents block on incoming message queues and become active when messages are received.

Non-concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [4, 6]. Every agent holds a counter of computation steps which it increments each time it performs a step. 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

```
– upon receiving message $msg$:
  1. LTC $\leftarrow$ max(LTC, msg.LTC)
  2. delay $\leftarrow$ $choose\_delay$
  3. msg.$delivery\_time$ $\leftarrow$ msg.LTC + delay
  4. $outgoing\_queue$.add(msg)
  5. $deliver\_messages$
– when there are no incoming messages and all agents are idle
  1. LTC $\leftarrow$ $outgoing\_queue$.first_msg.LTC
  2. $deliver\_messages$
– deliver_messages
  1. foreach (message m in outgoing queue)
  2.    if ($m.delivery\_time \leq$ LTC)
  3.       deliver(m)
```

Fig. 1. The Mailer algorithm

reporting the cost of the search as the largest counter held by some agent at the end of the search, a non-concurrent measure of search effort is achieved (see [4]).

On systems with message delays, the situation is different. To introduce the problems of counting in the presence of message delays, let us start with the simplest possible algorithm. Synchronous backtracking (SBT) performs assignments sequentially, one by one and no two assignments are performed concurrently [**?**]. Consequently, the effect of message delay is very clear. The number of computation steps is not affected by message delay and the delay in every step of computation is the delay on the message that triggered it. Therefore, the total time of the algorithm run can be calculated as the total computation time, plus the total delay time of messages. In the presence of concurrent computation, the time of message delays must be added to the run-time of the algorithm *only if no computation was performed concurrently*. To achieve this goal, the simulator counts message delays in terms of computation steps and adds them to the accumulated run-time. Such additions are performed only for instances when no computation is performed. In other words, when the delay of a message causes all agents to wait, performing no computation.

In order to simulate message delays, all messages are passed by a dedicated $Mailer$ thread. The mailer holds a counter of non-concurrent computation steps performed by agents in the system. This counter represents the logical time of the system and we refer to it as the *Logical Time Counter* (LTC). Every message delivered by the mailer to an agent, carries the LTC value of its delivery to the receiving agent. An agent that receives a message updates its counter to the maximum value between the received LTC and its own value. Next, it performs the computation step, and sends its outgoing messages with the value of its counter, incremented by 1. The same mechanism can be used for computing computational effort, by counting non-concurrent constraints checks. Agents add to the counter values in outgoing messages the number of constraints checks performed in the current step of computation.

The mailer simulates message delays in terms of non-concurrent computation steps. To do so it uses the LTC, according to the algorithm presented in figure 1. Let us go over the details of the $Mailer$ algorithm, in order to understand the measurements performed by the simulator during run time.

When the mailer receives a message, it first checks if the LTC value that is carried by the message is larger than its own value. If so, it increments the value of the LTC (line 1). In line 2 a delay for the message (in number of steps) is selected. Here, different types of selection mechanisms can be used, from fixed delays, through random delays, to delays that depend on the actual load of the communication network. To achieve delays that simulate dependency on network load, for example, one can assign message delays that are proportional to the size of the outgoing message queue.

Each message is assigned a $delivery\_time$ which is the sum of the value of the message's LTC and the selected delay (in steps), and placed in the $outgoing\_queue$ (lines 3,4). The $outgoing\_queue$ is a priority queue in which the messages are sorted by $delivery\_time$, so that the first message is the message with the lowest $delivery\_time$. In order to preserve the third assumption from section 2, messages from agent A_i to agent A_j cannot be placed in the outgoing queue before messages which are already in the outgoing queue which were also sent from A_i to A_j. This property is essential to asynchronous backtracking which is not correct without it (cf. [2]). The last line of the $Mailer$'s code calls method $deliver\_messages$, which delivers all messages with $delivery\_time$ less or equal to the mailer's current LTC value, to their destination agents.

When there are no incoming messages, and all agents are idle, if the $outgoing\_queue$ is not empty (otherwise the system is idle and a solution has been found) the mailer increases the value of the LTC to the value of the $delivery\_time$ of the first message in the outgoing queue and calls $deliver\_messages$. This is a crucial step of the simulation algorithm. Consider the run of a synchronous search algorithm. For *Synchronous Backtracking* (SBT) [13], every delay needs the mechanism of updating the Mailer's LTC (line 1 of the second function of the code in figure 1). This is because only one agent is computing at any given instance, in synchronous backtrack search.

The non-concurrent run time reported by the algorithm, is the largest LTC value that is held by any agent at the end of the algorithm's run. By incrementing the LTC only when messages carry LTCs with values larger than the mailer's LTC value, steps that were performed concurrently are not counted twice. This is an extension of Lamport's logical clocks algorithm [4], as proposed for DisCSPs by [6], and extended here for message delays.

A similar description holds for evaluating the algorithm run in non-concurrent constraints checks. In this case the agents need to extend the value of their LTCs by the number of constraints checks they actually performed in each step. This enables a concurrent performance measure that incorporates the computational cost of the local step, which might be different in different algorithms. It also enables to evaluate algorithms in which agents perform computation which is not triggered or followed by a message.

3.1 Adjusting the measuring method for dynamic order algorithms

In asynchronous backtracking with dynamic agent ordering [17] as in the Asynchronous Weak Commitment search algorithm, agents hold in their *Agent Views* assignments of both higher and lower priority agents. The agents check their current assignment only against assignments of agents with higher priority according to the current order. However, since the priority order is dynamic, an assignment of a lower priority agent which is currently irrelevant, may become relevant as a result of a change in the order of priorities, thus such lower priority assignments are not discarded from the agent's *Agent View*. The agents performing asynchronous backtracking with dynamic ordering (ABT_DO) or Asynchronous Weak Commitment (AWC), send their assignments to all their neighbors (and not only to their current lower priority neighbors) for the same reason [13, 17].

Messages which carry the assignments of lower priority agents to higher priority agents do not trigger immediate computation since the assignment in the received message cannot rule out the local assignment even if they are in conflict.

A small change in the agents actions would adjust the measuring method of $AMDS$ presented above for counting non-concurrent logic steps to deal with messages which do not trigger immediate computation, and their data is stored for later use. In order to preserve the concept of *non-concurrent* logic steps, for every message received, before updating the local *Logic Time Counter* (LTC) the agent must make sure that the computation performed in order to produce the data carried by the message *could not have been performed* concurrently with the steps of computation it is about to perform. Another way to look at it is to ask whether the computation steps about to be performed could have been performed if the message carrying the corresponding data was delayed. This can be done by the agents by delaying the update of their LTC in cases where the received LTC is larger. Instead the agents store the data in the received message together with the corresponding LTC. When the stored data is first used for computation, the corresponding LTC is compared with the local LTC and the last is updated with the largest among the two.

4 Validity of the $AMDS$

The validity of the proposed simulation algorithm can be established in two steps. First, its correspondence to runs of a *Synchronous (cycle-counting) Simulator* is presented. Understanding the nature of this correspondence, enables to show that a corresponding synchronous cycle simulator cannot measure concurrent delayed steps and the $AMDS$ is necessary.

In a *Synchronous Cycle Simulator* (SCS) [13], each agent can read all messages that were sent to it in the previous cycle and perform a single computation step. The computation is followed by the sending of messages (which will be received in the next cycle). Agents can be idle in some cycles, if they do not receive a message which triggers a computation step. The cost of the algorithm run, is the number of synchronous cycles performed until a solution is found or a non solution is declared (see [13]). Message delay can be simulated in such a synchronous simulator by delivering messages to agents several cycles after they were sent. Our first step is to show the correspondence of $AMDS$ and an SCS.

Theorem 1. *Any run of $AMDS$ can be simulated by a* Synchronous Cycle Simulator *(SCS). Each cycle c_i of the SCS corresponds to an LTC value of $AMDS$.*

Proof. Every message m sent by an agent A_i to agent A_j, using the $AMDS$, can be assigned a value d which is the largest value between the LTC carried by m in the $AMDS$ run and the value of the LTC held by A_j when it receives m. Running a *Synchronous Cycle Simulator* (SCS) and assigning each message m with the value d calculated as described above, the message can be delivered to A_j in cycle d. The outcome of this special SCS is that every agent in every cycle c_i receives the exact messages as the agents in the corresponding $AMDS$ and the histories of all these messages are equivalent. In this context the meaning of equivalent histories of messages is that at each step, the message has the same list of senders/receivers, each recording its step number which is the same. This means that agents have the same knowledge about the other agents as the agents performing the corresponding steps in the $AMDS$ run. Assuming the algorithm is deterministic, each agent will perform the same computation and send the same messages. If the algorithm includes random choices the run can be simulated by recording $AMDS$ choices and forcing the same choice in the synchronous simulator run. □

The theorem demonstrates that for measuring the number of steps of computation, the asynchronous simulator is equivalent to a standard SCS *that does not wait for all agents* to complete their computation in a given cycle, in order to move to the next cycle. Message delays are simulated simply by the SCS delivering messages in delayed cycles.

The validity and importance of the asynchronous simulator can now be understood. Consider the important case where computational effort needs to be measured, in terms of constraints checks for example. At each cycle agents perform different amounts of computation, depending on the algorithm, on the arrival of messages, etc. The SCS has no way to "guess" the amount of computation performed by each agent in any given step or cycle. It therefore cannot deliver the resulting message in the correct cycle (one that matches the correct amount of computation and waiting). The natural way to incorporate the computational cost into the performance measure is to "clock" the simulator by CCs (for example). But this is equivalent to using the $AMDS$ as proposed in section 3.

5 Asynchronous Backtracking search algorithms

This study focuses on asynchronous backtracking algorithms. The algorithms compared are standard asynchronous backtracking (ABT) [13], asynchronous backtracking with dynamic agent ordering (ABT_DO) [17] and asynchronous weak commitment (AWC) [13]. These algorithms are described in the following subsections.

5.1 Asynchronous Backtracking

The $Asynchronous\ Backtrack\ algorithm$ (ABT) was presented in several versions over the last decade and is described here in accordance with the more recent papers [13, 2]. In the ABT algorithm, agents hold an assignment for their variables at all times, which is consistent with their view of the state of the system (i.e. their $Agent\_view$). When the agent cannot find an assignment consistent with its $Agent\_view$, it changes

```
– when received $(\textbf{ok?}, (x_j, d_j))$ do
  1. add $(x_j, d_j)$ to $agent\_view$;
  2. check_agent_view;end_do;

– when received $(\textbf{nogood}, x_j, nogood)$ do
  1. add nogood to nogood list;
  2. when $nogood$ contains an agent $x_k$ that is not its neighbor do
  3.    request $x_k$ to add $x_i$ as a neighbor,
  4.    and add $(x_k, d_k)$ to $agent\_view$; end_do;
  5. $old\_value \leftarrow current\_value$; check_agent_view;
  6. when $old\_value$ = $current\_value$ do
  7.    send (ok?, $(x_i, current\_value)$) to $x_j$ ; end_ do; end_do;
– procedure check_agent_view
  1. when $agent\_view$ and $current\_value$ are not consistent do
  2.    if no value in $D_i$ is consistent with $agent\_view$ then backtrack;
  3.    else select $d \in D_i$ where $agent\_view$ and $d$ are consistent;
  4.       $current\_value \leftarrow d$;
  5.       send (ok?,$(x_i, d)$) to $low\_priority\_neighbors$; end_if;end_do;
– procedure backtrack
  1. $nogood \leftarrow resolve\_Nogoods$;
  2. when $nogood$ is an empty set do
  3.    broadcast to other agents that there is no solution;
  4.    terminate this algorithm; end_do;
  5. select $(x_j, d_j)$ where $x_j$ has the lowest priority in nogood;
  6. send (nogood, $x_i$, $nogood$) to $x_j$;
  7. remove $(x_j, d_j)$ from $agent\_view$; end_do;
  8. check_agent_view
```

Fig. 2. The ABT algorithm with full $Nogood$ recording

its view by eliminating a conflicting assignment from its $Agent\_view$ data structure and sends back a $Nogood$.

The ABT algorithm [13], has a total order of priorities among agents. Agents hold a data structure called $Agent\_view$ which contains the most recent assignments received from agents with higher priority. The algorithm starts by each agent assigning its variable, and sending the assignment to neighboring agents with lower priority. When an agent receives a message containing an assignment (an $ok?$ message [13]), it updates its $Agent\_view$ with the received assignment and if needed, replaces its own assignment, to achieve consistency. Agents that reassign their variable, inform their lower priority neighbors by sending them $ok?$ messages. Agents that cannot find a consistent assignment, send the inconsistent tuple in their $Agent\_view$ in a backtrack message (a $Nogood$ message [13]). The $Nogood$ is sent to the lowest priority agent in the inconsistent tuple, and its assignment is removed from their $Agent\_view$. Every agent that sends a $Nogood$ message, makes another attempt to assign its variable with an assignment consistent with its updated $Agent\_view$.

Agents that receive a $Nogood$, check its relevance against the content of their $Agent\_view$. If the $Nogood$ is relevant, the agent stores it and tries to find a consistent assignment. In any case, if the agent receiving the $Nogood$ keeps its assignment, it informs the $Nogood$ sender by re-sending it an $ok?$ message with its assignment. An agent A_i which receives a $Nogood$ containing an assignment of agent A_j which is not included in its $Agent\_view$, adds the assignment of A_j to it's $Agent\_view$ and sends a message to A_j asking it to add a link between them. In other words, A_j is requested to inform A_i about all assignment changes it performs in the future [2, 13].

The performance of ABT can be strongly improved by requiring agents to read all messages they receive before performing computation [13]. A formal protocol for such an algorithm was not published. The idea is not to reassign the variable until all the messages in the agent's 'mailbox' are read and the $Agent\_view$ is updated. This technique was found to improve the performance of ABT on the harder instances of randomly generated DisCSPs by a factor of 4 [15]. However, this property makes the efficiency of ABT dependent on the contents of the agent's mailbox in each step, i.e. on message delays (see section 6). 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 relevance of the messages it received up to this step.

Another improvement to the performance of ABT can be achieved by using the method for resolving inconsistent subsets of the $Agent\_view$, based on methods of dynamic backtracking. A version of ABT that uses this method was presented in [2]. In [15] the improvement of ABT using this method over ABT sending its full $Agent\_view$ as a $Nogood$ was found to be minor. In all the experiments in this paper a version of ABT which includes both of the above improvements is used. Agents read all incoming messages that were received before performing computation and $Nogoods$ are resolved, using the dynamic backtracking method [2].

The ABT algorithm is presented in figure 2 [13]. The first procedure is performed when an $ok?$ message is received. The agent adds the received assignment to its $Agent\_view$ and calls procedure $check\_agent\_view$. The second procedure is performed when a $Nogood$ is received. The $Nogood$ is stored (line 1), and a check is made whether it contains an assignment of a non neighboring agent. If so, the agent sends a message to the unlinked agent in order to establish a link between them and adds its assignment to its $Agent\_view$ (lines 2-4). Before calling procedure $check\_agent\_view$, the current value is stored (line 5). If for any reason the current value remains the same after calling $check\_agent\_view$, an $ok?$ message carrying this assignment is sent to the agent from whom the $Nogood$ was received (lines 6,7).

In procedure *check_agent_view* if the current value is not consistent with the $Agent\_view$ the agent searches its domain for a consistent value. If it does not find one, it calls procedure $backtrack$ (line 2). If there is a consistent value in its domain, it is placed as the $current\_value$ and $ok?$ messages are sent through all outgoing links (lines 3-5).

In procedure $backtrack$ the agent resolves its stored $Nogoods$ and chooses the $Nogood$ to be sent (line 1). If the $Nogood$ selected is empty, the algorithm is terminated unsuccessfully (lines 2-4). In other cases the agent sends the $Nogood$ to the agent with the lowest priority whose assignment is included in the $Nogood$, removes that assignment from the $Agent\_view$ and calls $check\_agent\_view$.

5.2 ABT with Dynamic Ordering (ABT_DO)

For simplicity of presentation we assume that agents send **order** messages to all lower priority agents. In the more realistic form of the algorithm, agents send **order** messages only to their lower priority *neighbors*.

Each agent in ABT_DO holds a $Current\_order$ which is an ordered list of pairs. Every pair includes the ID of one of the agents and a counter. Each agent can propose a new order for agents that have lower priority, each time it replaces its assignment. An agent A_i can propose an order according to the following rules:

1. Agents with higher priority than A_i and A_i itself, do not change priorities in the new order.
2. Agents with lower priority than A_i, in the current order, can change their priorities in the new order but not to a higher priority than A_i itself.

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

1. The counters of agents with higher priority than A_i, according to the $Current\_order$, are not changed.
2. The counter of A_i is incremented by one.
3. The counters of agents with lower priority than A_i in the $Current\_order$ are set to zero.

Consider an example in which agent A_2 holds the following $Current\_order$:
$(1,4)(2,3)(3,1)(4,0)(5,1)$. There are 5 agents $A_1...A_5$ and they are ordered according to their IDs from left to right. After replacing its assignment it changes the order to:
$(1,4)(2,4)(4,0)(5,0)(3,0)$. In the new order, agent A_1 which had higher priority than A_2 in the previous order keeps its place and the value of its counter does not change. A_2 also keeps its place and the value of its counter is incremented by one. The rest of the agents, which have lower priority than A_2 in the previous order, change places as long as they are still located lower than A_2. The new order for these agents is A_4, A_5, A_3 and their counters are set to zero.

In ABT, agents send **ok?** messages to their neighbors whenever they perform an assignment. In ABT_DO, an agent can choose to change its $Current\_order$ after changing its assignment. If that is the case, beside sending **ok?** messages an agent sends **order** messages to all lower priority agents. The **order** message includes the agent's new $Current\_order$. An agent which receives an **order** message must determine if the received order is more updated than its own $Current\_order$. It decides by comparing the time-stamps lexicographically. Since orders are changed according to the above rules, every two orders must have a common prefix of the agents IDs since the agent that performs the change does not change its own position and the positions of higher priority agents. In the above example the common prefix includes agents A_1 and A_2. Since the agent proposing the new order increases its own counter, when two different orders are compared, at lease one of the time-stamp counters in the common prefix is different between the two orders. The more up-to-date order is the one for which the first different counter in the common prefix is larger. In the example above, any agent which will receive the new order will know it is more updated than the previous order since the first pair is identical, but the counter of the second pair is larger.

When an agent A_i receives an order which is more up to date than its $Current\_order$, it replaces its $Current\_order$ by the received order. The new order might change the location of the receiving agent with respect to other agents (in the new $Current\_order$). In other words, one of the agents that had higher priority than A_i according to the old order, now has a lower priority than A_i or vise versa. Therefore, A_i rechecks the consistency of its current assignment and the validity of its stored $Nogoods$ according to the new order. If the current assignment is inconsistent according to the new order, the agent makes a new attempt to assign its variable. In ABT_DO agents send **ok?** messages to all constraining agents (i.e. their neighbors in the constraints graph). Although agents might hold in their $Agent\_views$ assignments of agents with lower priorities, according to their $Current\_order$, they eliminate values from their domain *only if they violate constraints with higher priority agents*.

A $Nogood$ message is always checked according to the $Current\_order$ of the receiving agent. If the receiving agent is not the lowest priority agent in the $Nogood$ according to its $Current\_order$, it sends the $Nogood$ to the lowest priority agent and sends an **ok?** message to the sender of the $Nogood$. This is a similar operation to that performed in standard ABT for any unaccepted $Nogood$.

Figures 3 and 4 present the code of asynchronous backtracking with dynamic ordering (ABT_DO).

When an **ok?** message is received (first procedure in Figure 3), the agent updates the $Agent\_view$ and removes inconsistent $Nogoods$. Then it calls **check_agent_view** to make sure its assignment is still consistent.

A new order received in an order message is accepted only if it is more up to date than the $Current\_order$ (second procedure of Figure 3). If so, the received order is stored and **check_agent_view** is called to make sure the current assignment is consistent with the higher priority assignments in the $Agent\_view$.

When a $Nogood$ is received (third procedure in Figure 3) the agent first checks if it is the lowest priority agent in the received $Nogood$, according to the $Current\_order$. If not, it sends the $Nogood$ to the lowest priority agent and an **ok?** message to the $Nogood$ sender (lines 1-3). If the receiving agent is the lowest priority agent it performs the same operations as in the standard ABT algorithm (lines 4-12).

Procedure **backtrack** (Figure 4) is the same as in standard ABT. The $Nogood$ is resolved and the result is sent to the lower priority agent in the $Nogood$, according to the $Current\_order$.

Procedure **check_agent_view** (Figure 4) is very similar to standard ABT but the difference is important (lines 5-9). If the current assignment is not consistent and must be replaced and a new consistent assignment is found, the agent chooses a new order as its $Current\_order$ (line 7) and updates the corresponding time-stamp. Next, **ok?** messages are sent to all neighboring agents. The new order and its time-stamp counters are sent to all lower priority agents.

```
when received (ok?, $(x_j, d_j)$ do:
1.  add $(x_j, d_j)$ to $agent\_view$;
2.  remove inconsistent $nogoods$;
3.  check_agent_view;

when received (order, $received\_order$) do:
1.  if ($received\_order$ is more updated than $Current\_order$)
2.     $Current\_order \leftarrow received\_order$;
3.     remove inconsistent nogoods;
4.     check_agent_view;

when received (nogood, $x_j, nogood$) do
1.  if ($nogood$ contains an agent $x_k$ with lower priority than $x_i$)
2.     send (nogood, $(x_i, nogood)$) to $x_k$;
3.     send (ok?, $(x_i, current\_value)$ to $x_j$;
4.  else
5.     if ($nogood$ consistent with $\{Agent\_view \cup current\_assignment\}$ )
6.        store $nogood$;
7.        if ($nogood$ contains an agent $x_k$ that is not its neighbor)
8.           request $x_k$ to add $x_i$ as a neighbor;
9.           add $(x_k, d_k)$ to $agent\_view$;
10.       check_agent_view;
11.    else
12.          send (ok?, $(x_i, current\_value)$) to $x_j$;
```

Fig. 3. The ABT_DO algorithm (first part)

5.3 Asynchronous Weak Commitment search

The Asynchronous Weak Commitment (AWC) search algorithm presented in [12] was constructed to increase the efficiency of the ABT algorithm. The major difference between AWC and standard ABT is that the priority order among agents is dynamic in AWC. An agent that cannot find a consistent assignment with it's $Agent\_view$, beside sending a $Nogood$, changes it's priority to be higher than all other agents [12].

In AWC, as in ABT_DO, **ok?** messages must be sent by agents to all their neighbors in the constraints network, not just the agents with lower priority. **ok?** messages must carry the agent's current priority, since the priorities change, and the other agents relate to the message received by comparing the received priority, with their own.

Unlike ABT_DO, in case of a backtrack operation, $Nogoods$ are sent to all agents whose assignment is included in the $Nogood$. Agents store all $Nogoods$ they receive. Agents also hold a list of the $Nogoods$ they have already sent to avoid sending the same $Nogood$ again. An exponential size $Nogood$ list is needed. This of course means that traversing the $Nogood$ list requires exponential computational cost.

The expected advantage of the AWC algorithm over ABT stems from its dynamic ordering of variables. AWC is more flexible than ABT_DO since its completeness is

```
procedure **check_agent_view**
1. if($current\_assignment$ is not consistent with all
         higher priority assignments in $agent\_view$)
2.    if(no value in $D_i$ is consistent with all higher priority
            assignments in $agent\_view$)
3.       backtrack;
4.    else
5.       select $d \in D_i$ where $agent\_view$ and $d$ are consistent;
6.       $current\_value \leftarrow d$;
7.       $Current\_order \leftarrow$ choose_new_order
8.       send (ok?,$(x_i, d)$) to $neighbors$;
9.       send (order,$Current\_order$) to lower priority agents;

procedure backtrack
1. $nogood \leftarrow$ resolve_inconsistent_subset;
2. if ($nogood$ is empty)
3.    broadcast to other agents that there is no solution;
4.    stop;
5. select $(x_j, d_j)$ where $x_j$ has the lowest priority in nogood;
6. send (nogood, $x_i$, $nogood$) to $x_j$;
7. remove $(x_j, d_j)$ from $agent\_view$;
8. remove all $Nogoods$ containing $(x_j, d_j)$;
9. check_agent_view;
```

Fig. 4. The ABT_DO algorithm(second part)

achieved by storing a complete list of $Nogoods$. Thus, its reordering is not restricted by the structure of the search tree. The main heuristic idea of $AWC's$ reordering is to move an agent A_i, which cannot assign its variable due to conflicting assignments of agents with higher priority, to the head of the priority order. This is expected to force the agents with the conflicting assignments to check for a value assignment in their domain, consistent with the assignment of A_i [12].

Figure 5 presents the **check_agent_view** and **backtrack** procedures of AWC (the other procedures are similar to standard ABT). Procedure **check_agent_view** is very similar to that of the dynamic ordered ABT (ABT_DO). The consistent asignment is checked to be consistent only against the assigtnments in the $Agent\_view$ which have higher priority. However, a check must be made that it does not violate any of the stored $Nogoods$. Once a consistent assignment is found, it is sent to all the agent's neighbors. In procedure **backtrack** there are two major difference from ABT. First the produced $Nogood$ is sent to all the agents whose assignment is included in the $Nogood$ (and not just to the last one). Second, the agent changes its priority to the highest one, before attemting to find a consistent assignment.

```
procedure **check_agent_view**
1. **when** $agent\_view$ and $current\_value$ are not consistent
2.    **if** (no value in $D_i$ is consistent with $agent\_view$)
3.       **backtrack**;
4.    **else**
5.       select $d \in D_i$ where $agent\_view$ and $d$ are consistent;
6.       $current\_value \leftarrow d$;
7.       send $($**ok?**$,(x_i, d))$ to $neighbors$;

procedure **backtrack**
1. $nogood \leftarrow$ **resolve_inconsistent_subset**;
2. **if** ($nogood$ is empty)
3.    broadcast to other agents that there is no solution;
4.    **stop**;
5. send (**nogood**, $x_i$, $nogood$) to all agents in $nogood$;
6. $priority \leftarrow maxpriorityinagent\_view + 1$
7. select $d \in D_i$ where $agent\_view$ and $d$ are consistent;
8. $current\_value \leftarrow d$;
9. send $($**ok?**$,(x_i, d))$ to $neighbors$;
```

Fig. 5. The AWC algorithm

6 Experimental evaluation

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. [7, 10]). The experiments were conducted on networks with 15 Agents ($n = 15$) and 10 values ($k = 10$). Two density parameters were used, $p_1 = 0.4$ and $p_1 = 0.7$. The value of p_2 was varied between 0.1 to 0.9. This creates problems that cover a wide range of difficulty, from easy problem instances to instances that take several CPU minutes to solve. For every pair (p_1,p_2) in the experiments we present the average over 50 randomly generated instances.

In order to evaluate the algorithms, two measures of search effort are used. One counts the number of non-concurrent constraint checks ($NCCCs$) [6, 16], to measure computational cost. This measures the combined path of computation, from beginning to end, in terms of constraint checks. The other measure used is the communication load, in the form of the total number of messages sent [5]. In order to evaluate the number of non-concurrent CCs including message delays, the simulator described in section 3 is used.

In the first set of experiments the impact of message delay was tested on the ABT algorithm with and without dynamic agent ordering. Figure 6 presents the number of non-concurrent constraints checks performed by ABT and dynamic ordered ABT on

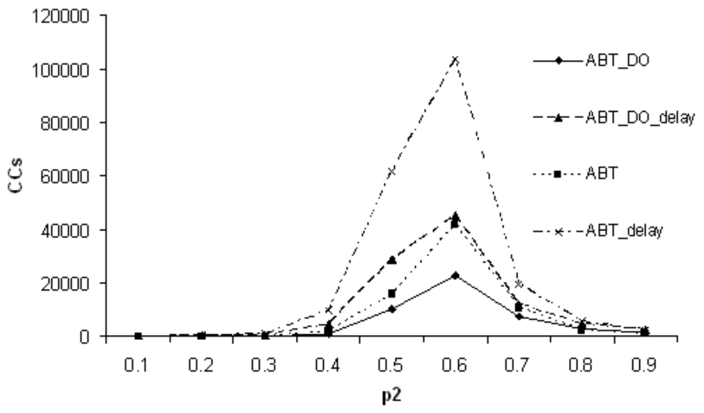

Fig. 6. Non-concurrent constraint checks performed by ABT and ABT_DO with and without message delays ($p_1 = 0.4$)

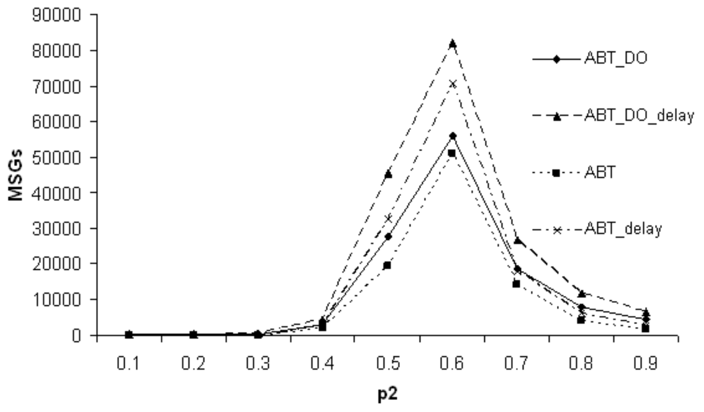

Fig. 7. Number of messages sent by ABT and ABT_DO with and without message delays ($p_1 = 0.4$)

systems with optimal communication (i.e. with no message delays) and on systems with random message delays between 50 and 100 CCs. It is apparent that the impact of meaasge delays on standard ABT is larger than on dynamically ordered ABT. Figure 7 presents the total number of messages sent by the agents performing the algorithms. The effect of message delays is similar on both algorithms. The number of messages increases by about 30%.

Figures 8 and 9 present similar results for more dense $DisCSPs$ ($p_1 = 0.7$). The factor of deterioration in the presence of message delays is similar to the factor in sparce $DisCSPs$.

In the second set of experiments, the well known AWC algorithm was evaluated on systems with optimal communication and on systems with random message delays.

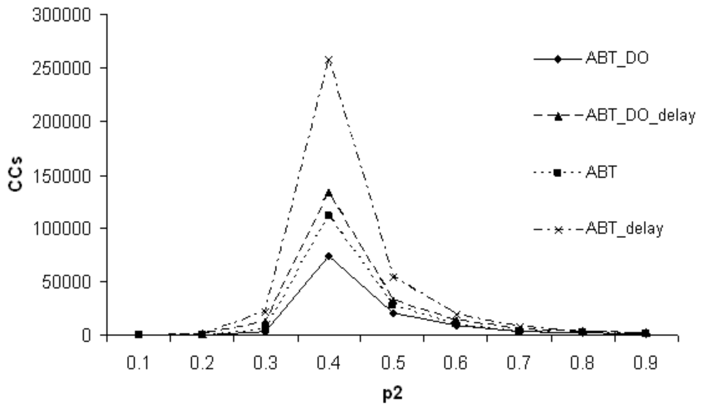

Fig. 8. Non-concurrent constraint checks performed by ABT and ABT_DO with and without message delays ($p_1 = 0.7$)

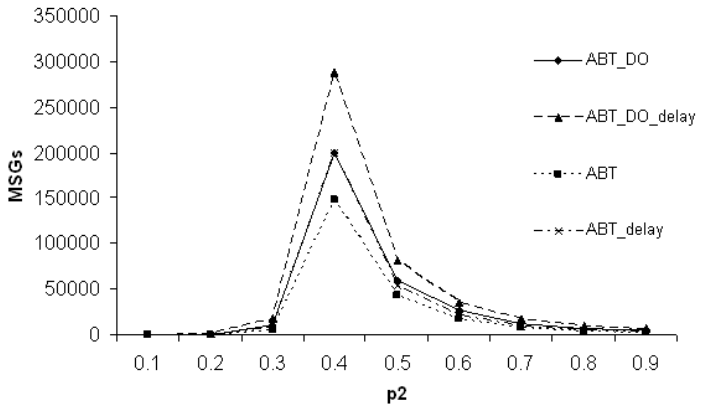

Fig. 9. Number of messages sent by ABT and ABT_DO with and without message delays ($p_1 = 0.7$)

Experiments were performed on smaller systems with 10 agents since AWC does not complete its runs in a reasonable time for larger problems in the presence of message delays.

Figure 10 presents the number of non-concurrent constraints checks performed by AWC on sparse systems ($p_1 = 0.4$). The factor of deterioration in $NCCCS$ for AWC is smaller than the factor for ABT and closer to the factor of deterioration for ABT_DO. However, in the case of network load, as presented in Figure 11, the factor of deterioration in the presense of message delays is much larger than for both versions of ABT.

Figures 12 and 13 present similar results for dense $DisCSPs$ ($p_1 = 0.7$).

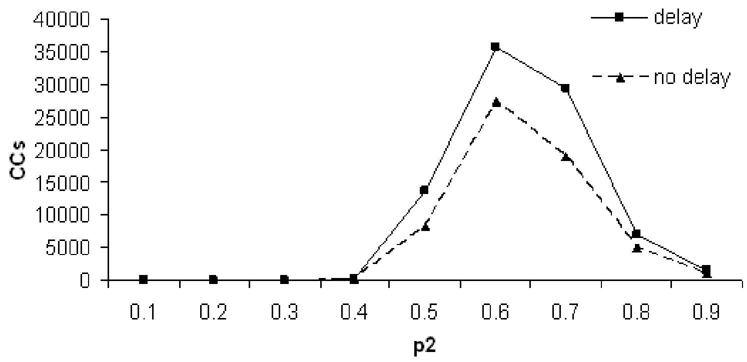

Fig. 10. Non-concurrent constraint checks performed by AWC with and without message delays ($p_1 = 0.4$)

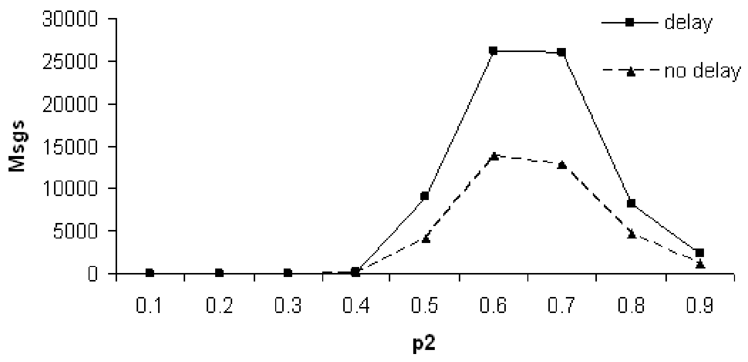

Fig. 11. Number of messages sent by AWC with and without message delays ($p_1 = 0.4$)

7 Discussion

Two sets of experiments to investigate the effect of message delays on the performance of Asynchronous Backtracking algorithms for $DisCSPs$ were performed.

In order to simulate message delays and include their impact in the experimental results, an asynchronous simulator which delivers messages to agents according to a logical time counter (LTC) of non-concurrent steps of computation (or non-concurrent constraints checks) was introduced. When computing logical time, the addition of message delay to the total cost occurs only when no concurrent computation is performed.

While in systems with perfect communication, where there are no message delays, the number of synchronous steps of computation (on a synchronous simulator) is a good measure of the time of the algorithm run, the case is different on realistic systems with message delays. The number of non-concurrent constraints checks has to take delays

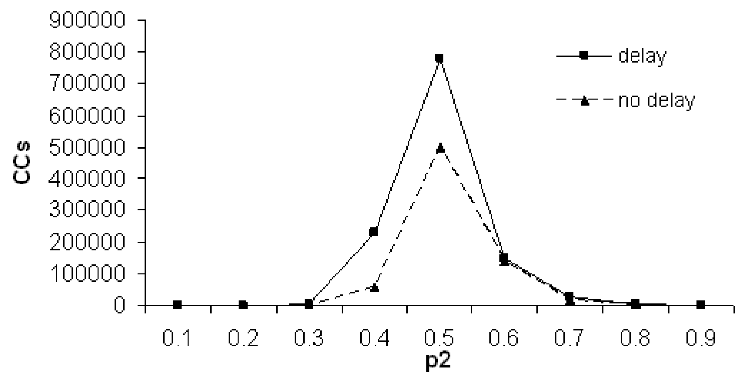

Fig. 12. Non-concurrent constraint checks performed by AWC with and without message delays $(p_1 = 0.7)$

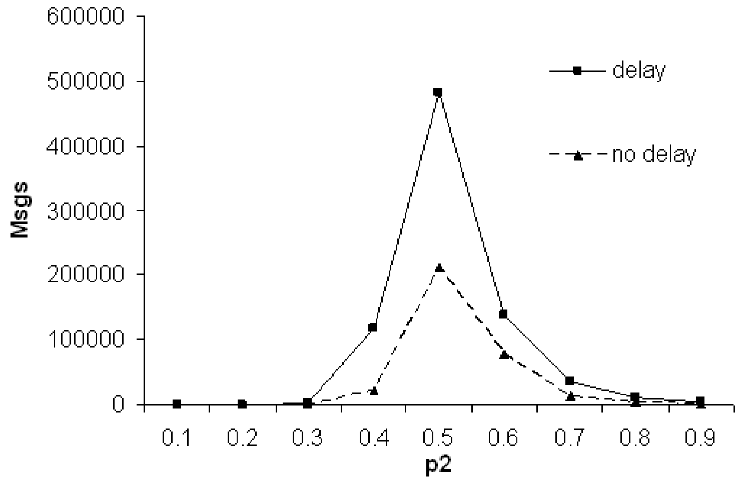

Fig. 13. Number of messages sent by AWC with and without message delays $(p_1 = 0.7)$

into account. When the number of non-concurrent CCs is calculated, it reveals a large impact of message delay on the performance of asynchronous backtracking algorithms..

In order to adjust the non-concurrent computational effort counting method of [6], for algorithms with dynamic ordering in which not every message triggers computation, the agents of the simulator store the information they receive and relate to the counters which represent their corresponding sending time, only when the data carried by the message is first used for computation.

In asynchronous backtracking, agents perform assignments asynchronously. As a result of random message delays, some of their computation can be irrelevant due to inconsistent $Agent\_views$ while the updating message is delayed. This can explain the large impact of message delays on the computation performed by ABT in our experiments and in a former study [1].

In terms of network load, the results of section 6 show that asynchronous backtracking puts a heavy load on the network, which grows in the case of message delays. The number of messages sent by the asychronous weak commitment algorithm (AWC) is larger than in ABT. This can be explained due to the fact that the number of messages sent in every step by AWC is larger than in ABT. Therefore, increase in the number of steps has a lager impact on the network load.

8 Conclusions

A study of the impact of message delay on the performance of DisCSP search algorithms was presented. A method for simulating logical time, in logical units such as non-concurrent steps of computation or non-concurrent constraint checks, has been introduced. The number of non-concurrent constraints checks takes into account the impact of message delays on the actual runtime of DisCSP algorithms. The impact of mesage delays on asynchronous backtracking, (ABT), is large. Both the computational effort and the load on the network grow by a large factor, This strengthens the results of [9, 1].

The effect of message delay on ABT with dynamic ordering is smaller but still significant. The runtime performance of the AWC algorithm reacts similarlly to ABT with dynamic ordering in the presence of message delays. However, it imposes a larger load on the network.

Acknowledgment: The authors wish to thank Moshe Zazone for programming and performing the experiments with the AWC algorithm.

References

[1] R. Bejar, C. Domshlak, C. Fernandez, , K. Gomes, B. Krishnamachari, B.Selman, and M.Valls. Sensor networks and distributed csp: communication, computation and complexity. *Artificial Intelligence*, 161:1-2:117–148, January 2005.

[2] C. Bessiere, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161:1-2:7–24, January 2005.

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

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

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

[6] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.

[7] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[8] M. C. Silaghi. *Asynchronously Solving Problems with Privacy Requirements*. PhD thesis, Swiss Federal Institute of Technology (EPFL), 2002.

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

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

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

[12] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, pages 88 – 102, Cassis, France, 1995.

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

[14] M. Yokoo, K. Hirayama, and K. Sycara. The phase transition in distributed constraint satisfaction problems: First results. In *Proc. CP-2000*, pages 515–519, Singapore, 2000.

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

[16] R. Zivan and A. Meisels. Concurrent dynamic backtracking for distributed csps. In *CP-2004*, pages 782–7, Toronto, 2004.

[17] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. In *CP-2005*, pages 32–46, Sigtes (Barcelona), Spain, 2005.