Message delay and DisCSP search algorithms ⋆

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. Due to the distributed nature of the problem, message delay can have unexpected effects on the behavior of distributed search algorithms on *Distributed constraint satisfaction problems* ($DisCSPs$). This has been recently shown in an experimental study of two asynchronous DisCSP algorithms [Fernandez et. al.2002]. To evaluate the impact of message delay on the run of DisCSP search algorithms, an *Asynchronous Message Delay Simulator* ($AMDS$) for $DisCSP$s which includes the cost of message delays is introduced. The number of steps of computation calculated by the $AMDS$ (or number of concurrent constraints checks) can serve as good performance measures, when messages are delayed.
The performance of three representative algorithms is measured on randomly generated instances of DisCSPs with several types of delays for messages.
Two measures of performance are used - concurrent computation time and network load. The performance of asynchronous backtracking deteriorates on systems with random message delays, for both measures. For synchronous algorithms, with delayed messages, time performance becomes worse then asynchronous backtracking, but the network load is not affected. Concurrent search algorithms, are affected very lightly by message delay with respect to both measures.

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 their 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 with variables that belong to different agents [Yokoo2000,Bessiere et. al.2001].

Search algorithms on DisCSPs are run concurrently by all agents and their performance must be measured in terms of distributed computation. Two measures are

⋆ Partially supported by the Lynn and William Frankel Center for Computer Science

commonly used to evaluate distributed algorithms - time, which is measured in terms of computational effort and network load [Lynch1997]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [Yokoo2000]). 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 concurrent constraints checks that agents perform ([Meisels et. al.2002,Silaghi2002]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [Lynch1997].

When instantaneous message arrival is assumed, steps of computation in a standard simulator can serve to measure the concurrent run-time of a DisCSP algorithm [Yokoo2000]. For an optimal communication network, in which messages arrive with no delay, one can also use the number of concurrent constraints checks (CCCs), for an implementation independent measure of concurrent run time [Meisels et. al.2002]. On realistic networks, in which there are variant message delays, the time of run cannot be measured simply by the steps of computation. Take for example Synchronous Backtracking [Yokoo2000]. 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).

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 in steps of computation or concurrent constraints checks, and simulates message delays accordingly. The $AMDS$ is described in detail in section 3. 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. Since the delay is measured in concurrent computation steps (or 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).

To demonstrate the behavior of DisCSP search algorithms in the presence of message delay, three algorithms are compared. Although the three chosen algorithms are similar in their run-time results on systems with no message delay they are very different from one another. The first, *Conflict based Back Jumping* (CBJ) [Zivan and Meisels2003] is a synchronous algorithm which performs pruning of its search space according to *Dynamic Backtracking* (DB) methods [Ginsberg1993,Zivan and Meisels2003]. The second is the *Asynchronous Backtracking* (ABT) algorithm in which agents perform assignments concurrently and asynchronously [Yokoo2000,Bessiere et. al.2001]. The third, *Concurrent Backtracking* [Zivan and Meisels2004] is a concurrent algorithm in which a dynamic number of independent search processes explore concurrently and asynchronously, non intersecting parts of the $DisCSP$ search space. The results presented in section 6 show the different impact of message delays on these three algorithms.

Distributed constraints satisfaction problems ($DisCSPs$) are presented briefly in section 2. A detailed introduction of the algorithm and method for simulating message

delays in $DisCSP$ search, and of the method of evaluating the run time of an algorithm, is presented in section 3. A proof of the validity of the simulating algorithm is presented in section 4. A description of the compared algorithms - synchronous Conflict based Backjumping (CBJ), Asynchronous Backtracking (ABT), and Concurrent Backtracking ($ConcBT$), is presented in section 5. Section 6 presents extensive experimental results, comparing all three algorithms on randomly generated $DisCSP$s with different types of message delays. A discussion of the new insights of the performance and on the advantages of these three algorithms, on different $DisCSP$ instances and communication networks, is 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 variables that belong to different agents (cf. [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 *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is a partial assignment 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 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 Distributed CSPs and are assumed to hold in the present study [Yokoo2000,Bessiere et. al.2001].

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.

Concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [Lamport1978,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, we achieve a measure of concurrent search effort that is similar to Lamport's logical time [Lamport1978].

On systems with message delays, the situation is more complex. For the simplest possible algorithm, Synchronous Backtrack (SBT) [Yokoo2000], 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 total algorithm time *only if no computation was performed concurrently*. To achieve this goal, the algorithm of the *Asynchronous Message-Delay Simulator* ($AMDS$) counts message delays in terms of computation steps and adds them to the accumulated run-time when no computation is performed concurrently.

In order to simulate message delays, all messages are passed by a dedicated $Mailer$ thread. The mailer holds a counter of 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. To compute the logical time that includes message delays, agents perform a similar computation to the one used when there are no message delays [Meisels et. al.2002]. An agent that receives a message updates its own LTC to the largest value between its own and the LTC on the message received. Then the agent performs the computation step, and sends its outgoing messages with the value of its LTC incremented by 1.

The mailer simulates message delays in terms of concurrent computation steps. To do so it uses its own (global) LTC, according to the algorithm presented in figure 1. Let

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

Fig. 1. The Mailer algorithm

us go over the details of the $Mailer$ algorithm, in order to understand the measurements performed by the $AMDS$ 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). This generates the value of the global clock (of the Mailer) which is the largest of all logical times of all agents. 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 current value of the Mailer'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 of 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 sent from A_i to A_j. This property is essential to asynchronous algorithms which are not correct without it (cf. [Bessiere et. al.2001]). 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) [Yokoo2000], 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 concurrent run time reported by the algorithm, is the largest LTC held by some agent at the end of the algorithm 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 [Lamport1978], as proposed for DisCSPs by [Meisels et. al.2002], and extended here for message delays.

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

4 Correctness of the $AMDS$

In order to prove the validity of the proposed measure simulation, its correspondence to runs of a *Synchronous Cycles Simulator* is presented. In a *Synchronous Cycle Simulator* [Yokoo2000], in every cycle each agent can read all messages that were sent to it in the previous cycle and perform a single computation step which can be 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 [Yokoo2000]). Message delay can be simulated in such a synchronous simulator by delivering messages to agents some cycles after they were sent.

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

The proof of the theorem is immediate. Every message m sent by an agent A_i to agent A_j 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 received 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 the special SCS is that every agent in every cycle c_i will have the same knowledge about the other agents as the agents performing the matching steps in the $AMDS$ run. Assuming the algorithm is deterministic, the 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. To complete the proof of the theorem one needs to show the following Lemma.

Lemma 1. *At any cycle c_i of the synchronous simulator, the LTC values of all agents performing the matching steps in the $AMDS$ are equal to i.*

Proof: We prove Lemma 1 by induction. After performing step number one, all agents in $AMDS$ advance their LTC to one. Assuming the Lemma holds for $N-1$ cycles, all agents that are about to perform the Nth step, hold counters with values less or equal to $N-1$. The messages they will receive will carry the $delivery\_time$ LTC which is $N-1$. Since the agent's LTCs are updated to the largest between the received LTC and their own, after receiving the message and performing the next step of computation, their LTC value will be equal to N. □

The theorem demonstrates that for computing 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.

The most important advantage of the asynchronous simulator can now be described. When computational effort is computed, in terms of constraints checks for example, the SCS becomes useless. This is because at each cycle agents perform different amounts of computation, depending on the algorithm, on arrival of messages, etc. The simulator does not "know" the amount of computation performed by each agent and, therefore, cannot move the resulting message in the correct cycle (one that matches the correct amount of computation and waiting). The natural way to compute concurrent CCs is by using an asynchronous simulator, the $AMDS$ as proposed in section 3

5 The tested algorithms

In order to check the behavior of distributed search algorithms under message delays, the $AMDS$ is used to compare the run of three algorithms for solving $DisCSP$s. These algorithms represent three different families of algorithms:

- Synchronous algorithms represented by synchronous Conflict based Backjumping (CBJ) [Zivan and Meisels2003].
- Asynchronous Backtracking algorithms represented by ABT [Bessiere et. al.2001].
- Concurrent search algorithms represented by Concurrent Backtracking ($ConcBT$) [Zivan and Meisels2004].

In the following subsections the three representative algorithms are described. The performance of the algorithms is evaluated in section 6 and the impact of delayed messages on their performance is described. The relation of the impact of delayed messages on each of the algorithms and the properties of the algorithm's family, is discussed in section 7.

5.1 Conflict based Backjumping

The Synchronous Backtrack algorithm (SBT) [Yokoo2000], is a distributed version of chronological backtrack [Prosser1993]. SBT has a total order among all agents. Agents exchange a partial solution that we term *Current Partial Assignment* (CPA) which carries a consistent tuple of the assignments of the agents it passed so far. The first agent initializes the search by creating a CPA and assigning its variables on it. Every agent

that receives the CPA tries to assign its variable without violating constraints with the assignments on the CPA. If the agent succeeds to find such an assignment to its variable, it appends the assignment to the tuple on the CPA and sends it to the next agent. Agents that cannot extend the consistent assignment on the CPA, send the CPA back to the previous agent to change its assignment, thus perform a chronological backtrack. An agent that receives a CPA in a backtrack message removes the assignment of its variable and tries to reassign it with a consistent value. The algorithm ends successfully if the last agent manages to find a consistent assignment for its variable. The algorithm ends unsuccessfully if the first agent encounters an empty domain.

The version of Conflict based Backjumping (CBJ) [Prosser1993] improves on simple synchronous backtrack (SBT) by using a method based on dynamic backtracking [Ginsberg1993,Bessiere et. al.2001]. In the distributed CBJ, when an agent removes a value from its variable's domain, it stores the eliminating explanation ($Nogood$), i.e. the subset of the CPA that caused the removal. As in the corresponding version of asynchronous backtrack [Bessiere et. al.2001], when a backtrack operation is performed the agent resolves its $Nogoods$ creating a conflict set which is used to determine the culprit agent to which the backtrack message will be sent. The resulting synchronous algorithm has the backjumping property (i.e. CBJ) [Ginsberg1993]. When the CPA is received again, values whose eliminating $Nogoods$ are no longer consistent with the partial assignment on the CPA are returned to the agents' domain.

5.2 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 [Yokoo2000,Bessiere et. al.2001]. 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 its view by eliminating a conflicting assignment from its $Agent\_view$ data structure and sends back a $Nogood$.

The Asynchronous Backtrack algorithm ABT [Yokoo2000], 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 [Yokoo2000]), 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 [Yokoo2000]). 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 con-

sistent 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 [Bessiere et. al.2001]. 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 [Yokoo2000].

The performance of ABT can be strongly improved by requiring agents to read all messages they receive before performing computation [Yokoo2000]. 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 [Zivan and Meisels2003]. 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 [Ginsberg1993]. A version of ABT that uses this method was presented in [Bessiere et. al.2001]. In [Zivan and Meisels2003] 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.

5.3 Concurrent Backtracking

The $ConcBT$ algorithm [Zivan and Meisels2004] performs multiple concurrent backtrack searches on disjoint parts of the *DisCSP* search-space. Each agent holds the data relevant to its state on each sub-search-space in a separate data structure which is termed *Search Process (SP)*. Agents in the $ConcBT$ algorithm pass their assignments to other agents on a *CPA* (Current Partial Assignment) data structure. Each *CPA* represents one search process, and holds the agents current assignments in the corresponding search process. An agent that receives a *CPA* tries to assign its local variable with values that are not conflicting with the assignments on the *CPA*, using the current domain in the *SP* related to the received *CPA*. The uniqueness of the *CPA* for every search space ensures that assignments are not done concurrently in a single sub-search-space.

Exhaustive search processes which scan heavily backtracked search-spaces, can be split dynamically. Each agent can generate a set of CPAs that split the search space of a CPA that passed through that agent, by splitting the domain of its variable. Agents can perform splits independently and keep the resulting data structures (*SP*s) privately. All other agents need not be aware of the split, they process all $CPAs$ in exactly the same

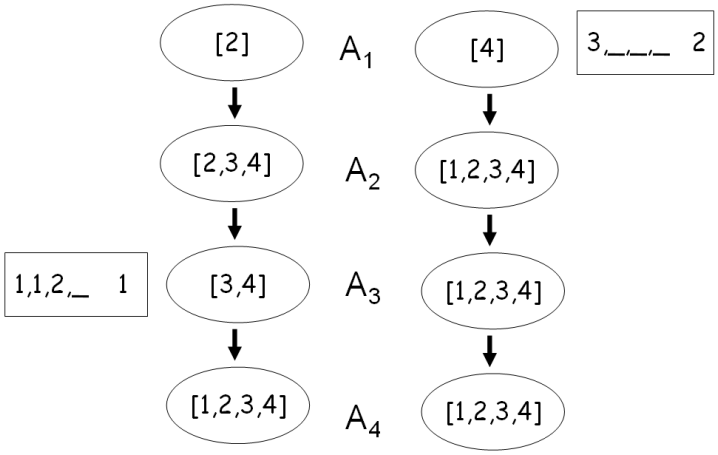

Fig. 2. ConcBT with two CPAs

manner (see [Zivan and Meisels2004] for a detailed explanation). *CPA*s are created either by the Initializing Agent (*IA*) at the beginning of the algorithm run, or dynamically by any agent that splits an active search-space during the algorithm run. A heuristic of counting the number of times agents pass the *CPA* in a sub-search-space (without finding a solution), is used to determine the need for re-splitting of that sub-search-space. This generates a mechanism of load balancing, creating more search processes on heavily backtracked search spaces.

A backtrack operation is performed by an agent which fails to find a consistent assignment in the search-space corresponding to the partial assignment on the *CPA*. Agents that have performed dynamic splitting, have to collect all of the returning CPAs, of the relevant SP, before performing a backtrack operation.

Figure 2 presents an example of a DisCSP, searched concurrently by two synchronous processes represented by two CPAs, CPA_1 and CPA_2. Each of the four agents A_1 to A_4, holds two SPs. Only the current domains of the SPs are shown in Figure 2. The domains on the left represent the state after 3 assignments to CPA_1. The domains on the right of figure 2 represent the state after the first assignment to CPA_2.

Agent A_1 has assigned the value 1 on CPA_1 and the value 3 on CPA_2. The values that are left in each of its domains are 2 in SP_1 and 4 in SP_2. The two CPAs are traversing non intersecting sub search spaces in which CPA_1 is exploring all tuples beginning with 1 or 2 for agent A_1, and CPA_2 all tuples beginning with 3 or 4. CPA_1 is depicted on the LHS of figure 2 and CPA_2 is on the top RHS. Each CPA has its ID on its right.

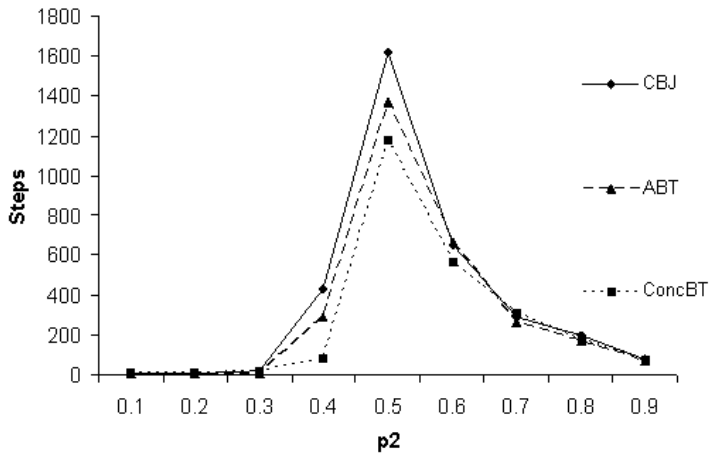

Fig. 3. Concurrent steps of computation with no message delays

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. [Prosser1996]). Experiments were conducted on networks with 10 variables ($n = 10$) and 10 values ($k = 10$). All instances were created with density parameter $p_1 = 0.7$. 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.

In order to evaluate the algorithms, two measures of search effort are used. One counts the number of concurrent steps of computation [Lynch1997,Yokoo2000], to measure computational cost. The other measures communication load in the form of the total number of messages sent [Lynch1997]. Concurrent steps of computation are counted by a method similar to that of [Lamport1978,Meisels et. al.2002]. In order to evaluate the logical time (including message delays) of the algorithm, in steps of computation, we use the simulator as described in section 3.

In the first set of experiments the three algorithms are compared without any message delay. The results presented in figure 3 show that the numbers of steps of computation that the three algorithms perform are very similar, on systems with no message delays. ABT performs slightly less steps than CBJ and $ConcBT$ performs slightly better than ABT. However, when it comes to network load, the results in figure 4 show that for the harder problem instances, agents in ABT send *six times more messages* than sent by agents in CBJ and more than twice the number of messages sent by agents in $ConcBT$.

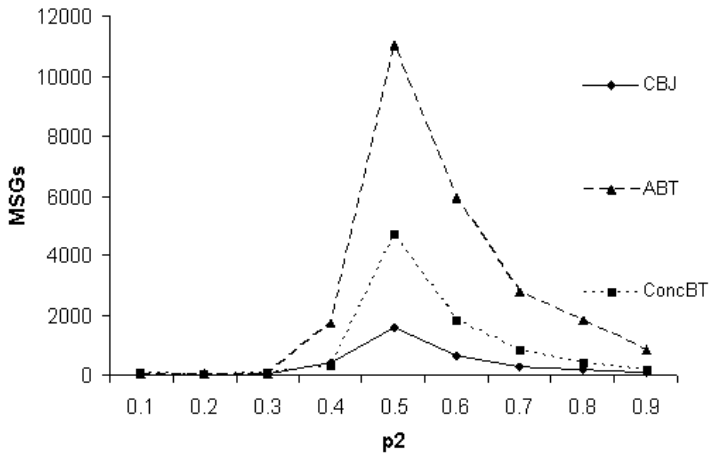

Fig. 4. Total number of messages with no message delays

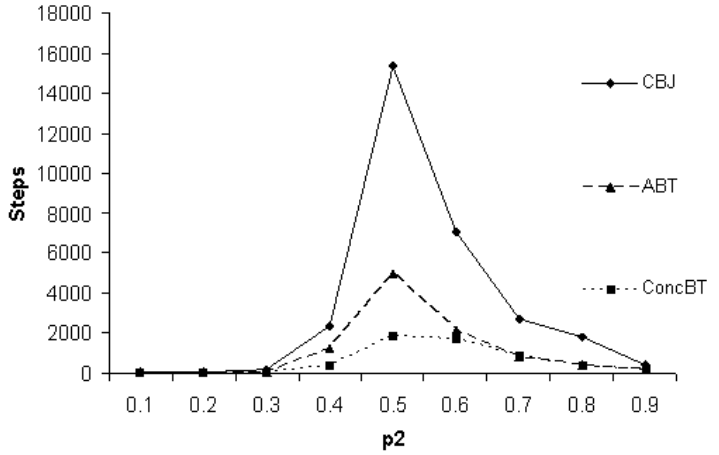

Fig. 5. Logical number of concurrent steps with random message delays

In the second set of experiments, the simulator's $Mailer$ delayed messages randomly for 5-10 steps (as described in section 3).

Figure 5 presents the results of logical time, counted in concurrent steps, for random message delays. It is clear in figure 5 that even though message delays do not affect the number of concurrent steps performed by agents in CBJ, when message delay is correctly counted, CBJ is affected the most. The number of steps performed by ABT in the presence of delays, grows by a large factor. This is expected, since agents are more likely to respond to a single message, instead of all the messages sent in the former (ideal) cycle of computation. Messages in asynchronous backtracking are many times

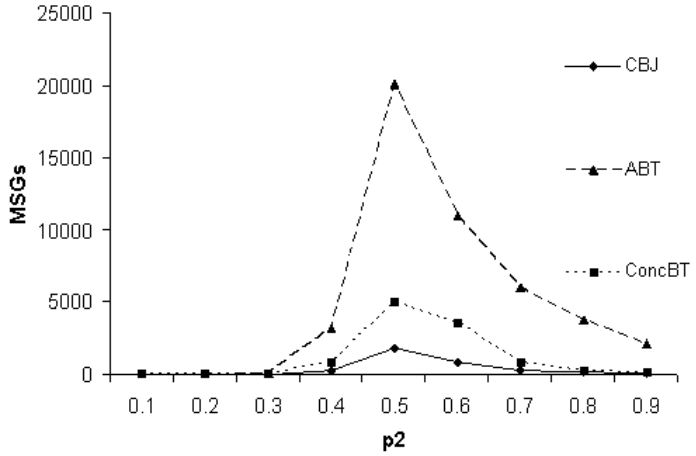

Fig. 6. Number of messages with random message delays

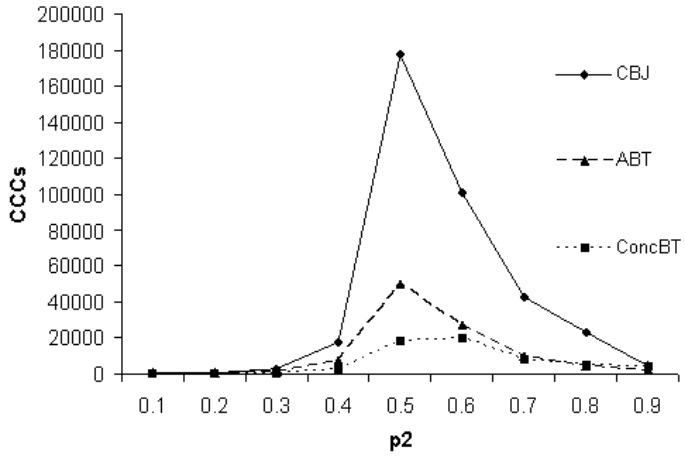

Fig. 7. Logical number of concurrent constraints checks with random message delays

conflicting. As a result, agents perform more unnecessary computation steps when responding to fewer messages in each cycle.This can explain a similar result for ABT, on a different set of problems [Fernandez et. al.2002]. The logical time performance of concurrent search process algorithms, is not strongly affected by message delay. For the harder problems $ConcBT$ performs less than half the steps of computation of ABT (see figure 5). Network load, for the same (delayed messages) experiments is presented in Figure 6. Both CBJ and $ConcBT$ send the same number of messages as in the case of no message delays. The number of messages sent by asynchronous backtracking increases dramatically. ABT sends almost twice as much messages in the presence of ran-

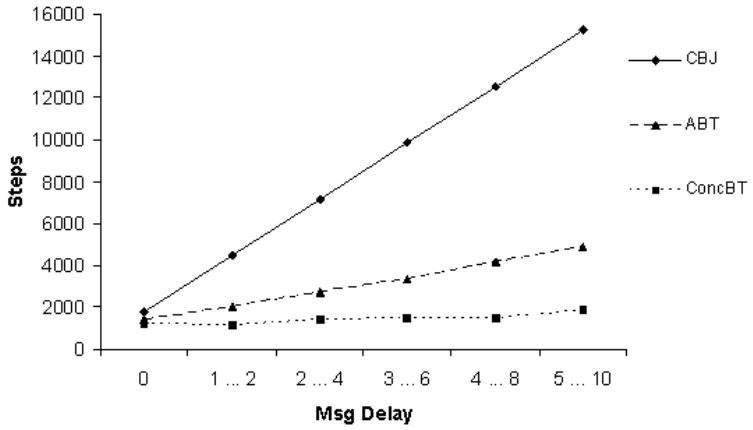

Fig. 8. Logical number of steps with different random message delays

dom message delays, than it sends in the case of no message delays (figure 6). Figure 7 presents the results for logical time that is counted in units of concurrent constraints checks. In this experiment the local computation is taken in to account. The delay for every message is chosen a random value between 50 to 150 constraints checks.

The last set of experiments tests the dependence of algorithm performance on the *amount of delay of messages*. All algorithms are run on the hardest problem instances (p_2 = 0.5) with an increasing amount of message delay. The different impact of random delays on the different algorithms is presented in figure 8. The number of steps of synchronous and of asynchronous backtracking grows with the size of message delay. In contrast, larger delays do not have an impact on the number of steps of concurrent search (Figure 8).

7 Discussion

A study of the impact of message delays on the behavior of $DisCSP$ search algorithms has been presented. Use was made of an asynchronous simulator that runs the $DisCSP$ algorithms with different types of message delays and measures performance in concurrent steps of computation. The logical number of steps/constraints-checks takes into account the impact of message delays on the actual runtime of $DisCSP$ algorithms. Three different algorithms for solving $DisCSPs$ were investigated.

In asynchronous backtracking, agents perform assignments asynchronously. As a result of message delay, 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 (cf. [Fernandez et. al.2002]). The results presented in section 6 strengthen the results reported by Fernandez et. al. [Fernandez et. al.2002], and do so for a larger family of random problems.

The impact of message delays on concurrent search algorithms is minor. This is very apparent in Figure 8, where the number of steps of computation is independent of the size of message delay for $ConcBT$.

To understand the robustness of $ConcBT$ to message delay imagine the following example. Consider the case where $ConcBT$ splits the search space multiple times and the number of CPAs is larger than the number of agents. In systems with no message delays this would mean that some of the CPAs are waiting in incoming queues, to be processed by the agents. This delays the search on the sub-search-spaces they represent. In systems with message delays, these queues are shortened due to later arrivals of $CPAs$. The net result is that agents are kept busy at all times, performing computation against consistent partial assignments. The results in section 6 demonstrate that the above possible description can be achieved.

In terms of network load, the results of the experimental investigation show that asynchronous backtrack puts a heavy load on the network, which doubles in the case of message delays. The number of messages sent, in both synchronous and concurrent algorithms, is much smaller than the load of asynchronous backtracking and is not affected by message delays.

References

[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*, pages 664-679, Ithaca NY USA, July, 2002.

[Ginsberg1993] M. L. Ginsberg Dynamic Backtracking. *Artificial Intelligence Research*, vol.1, pp. 25-46, 1993

[Lamport1978] 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 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.

[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.

[Silaghi2002] M.C. Silaghi Asynchronously Solving Problems with Privacy Requirements. *PhD Thesis*,Swiss Federal Institute of Technology (EPFL), 2002.

[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 Problem: 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 Meisels2003] R. Zivan and A. Meisels. Synchronous vs. Asynchronous search on DisCSPs. *Proc. EUMAS-03 1st European Workshop on Multi-agent Systems*, pp. 202-208, Oxford, December, 2003.

[Zivan and Meisels2004] R. Zivan and A. Meisels. Concurrent Backtrack Search on DisCSPs. *Proc. FLAIRS-04*, Miami Beach, May, 2004. (full version can be loaded from http://www.cs.bgu.ac.il/ zivanr)