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. 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 recently shown in experimental studies of asynchronous backtracking algorithms [1, 15].
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 non-concurrent number of 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. Two families of distributed search algorithms on DisCSPs are investigated. Algorithms that run a single search process, and multiple search processes algorithms. The two families of algorithms are described and associated with existing algorithms. The performance of three representative algorithms of these two families is measured on randomly generated instances of DisCSPs with delayed messages. The delay of messages is found to have a strong negative effect on single search process algorithms, whether synchronous or asynchronous. Multi search process algorithms, on the other hand, are affected very lightly by message delay.

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. [18, 17]). 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 [18, 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 [8]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [18]). 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 ([9, 13]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [8].

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 [18]. The number of these synchronous steps of computation in a standard simulator served to measure the non-concurrent run-time of a DisCSP algorithm [18]. 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 [7, 9].

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 [9]. 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) [18]. 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 [7] and uses constraints checks as a logical time unit [9]. 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 families 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, 15]. 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.

The present study of the behavior of distributed search algorithms on DisCSPs uses a selected set of three very different $DisCSP$ algorithms. All search algorithms on $DisCSPs$ can be divided into two families. Single search process algorithms (SPAs) and *concurrent* (multiple) *search* process algorithms *(CSAs)*. The only former experimental study of the performance of DisCSP algorithms compared two asynchronous single search algorithms [1].

The state of single process algorithms is defined by a *single tuple of assignments*, one for each agent. When this set of assignments is complete (containing assignments to all variables of all agents) and consistent, the SPA stops and reports a solution. A simple representation for the state of any *synchronous* SPA, like SBT [18] or CBJ [21], is a data structure that holds the *current partial assignment* of the search (CPA). Single search process algorithms can be asynchronous. In *asynchronous* backtracking (ABT) [18, 2], each agent holds its view of the current assignments of other agents in a single $Agent\_view$. When all agents are idle, all $Agent\_Views$ are consistent and a solution is reported [18, 2].

In concurrent search, multiple concurrent processes perform search on non intersecting parts of the global search space of a DisCSP ([14, 20, 6]). All agents in a CSA participate in every search process, since each agent holds some variables of the search space. Each agent holds the current domains of its variables, for each of the search processes. Messages of CSAs carry the IDs of their related search process and the agents use the corresponding current domains for consistent assignments. The concurrent backtracking algorithm ($ConcBT$), distributes among agents a dynamically changing number of search processes [22]. Agents generate and terminate search processes dynamically during the run of the $ConcBT$ algorithm [22]. The concurrent dynamic backtracking ($ConcDB$) algorithm incorporates dynamic backtracking to the concurrent performing search processes. As a result, one search procedure can reveal a dead end of another concurrent search procedure and terminate it [23].

In interleaved asynchronous backtracking, agents participate in multiple processes of asynchronous backtracking. Each agent keeps a separate $Agent\_view$ for each search process in $IDIBT$ [6]. The number of search processes is fixed by the first agent. The performance of concurrent asynchronous backtracking [14, 6] was tested and found to be ineffective for more than two concurrent search processes [6].

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 families of $DisCSP$ search algorithms - single process (SPAs) and concurrent search (CSAs). This is followed by a detailed description of the compared algorithms - synchronous SPA (CBJ), asynchronous backtracking (ABT), and concurrent search ($ConcDB$). The first two algorithms have appeared in different versions in the literature and the compared versions are the most up to date. Synchronous BT uses backjumping [21, 3] and asynchronous BT resolves $Nogoods$ [2]. The representative concurrent search algorithm is $ConcDB$ which was found to perform best in a recent study [23]. 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 performance and advantages of the families of 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 [4]. 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. [18, 17]). 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 [18].

The delay in delivering a message is assumed to be finite [18]. 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 [18, 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 [7, 9]. 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 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 [7]).

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

– **upon receiving message** msg:
  1. LTC ← max(LTC, msg.LTC)
  2. delay ← $choose\_delay$
  3. msg.$delivery\_time$ ← msg.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. deliver(m)

Fig. 1. The Mailer algorithm

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) [18], 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 [7], as proposed for DisCSPs by [9], 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.

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) [18], 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 [18]). 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. 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 Families of DisCSP search algorithms

Algorithms for solving DisCSPs can be divided into two families: single search process algorithms (SPAs) and multiple search process algorithms (MPAs). The general model of DisCSPs has variables owned by agents, who assign them values. The distinction between the two families of algorithms is in the number of concurrent assignments that agents maintain. In SPAs each agent can have no more than one assignment to its variable, at any single instance. In multiple process algorithms (MPAs), on the other hand, agents maintain multiple concurrent assignments to their variable. To give an example, synchronous backtracking (SBT) is a single process algorithm. During search, a single CPA carries the assignments of some of the agents. The other agents which are waiting for the message with assignments to arrive, are still unassigned. Therefore, each agent, in every step of the search, has either one assignment or none [18]. Asynchronous backtracking (ABT) is also a SPA. All the variables in ABT have exactly one assignment at each instant of its run [2].

To maintain two concurrent assignments in a DisCSP, think of the first agent as assigning two of its values to its variable. It then puts each assignment on a different message and initializes a backtracking process for each one. Each backtrack process

traverses all agents, not necessarily at the same order, to accumulate assignments to all variables of all agents. All agents receive eventually two messages. One message has the first assignment for the first agent and the other has the second assignment that the first agent performed. Agents that receive a message either add their compatible assignment to the partial assignment already on the message, or backtrack by sending the message back. All agents use a different current domain for each of the messages. It is easy to see that all agents react to the two messages in exactly the same way, assigning their variable on it or backtracking. This process stops when one of the messages accumulates a complete assignment and reports a solution, or when both messages return to the first agent and find no more values to assign. In this case the two-process algorithm reports failure.

Several single process DisCSP search algorithms have appeared in the literature in the last decade. Synchronous algorithms like synchronous backtrack (SBT) and conflict-based backjumping (CBJ) [18, 21]. Asynchronous algorithms like asynchronous backtracking (ABT), asynchronous aggregations search (AAS) and asynchronous forward-checking (AFC) [15, 2, 10]. In contrast, only few multiple process DisCSP search algorithms appear in the literature [14, 6, 22]. The concurrent dynamic backtracking algorithm ($ConcDB$), with dynamic splitting of search processes, will be the representative of this family in the present study. $ConcDB$ incorporates dynamic splitting, generating a variable number of search processes. Furthermore, the search processes cooporate inorder to detect and terminate invalid active search processes [23]. Multiple search process versions of asynchronous backtracking were found not to improve for more than 2 concurrent processes [6]

In the following subsections three representative algorithms of the above two families are described. Two single-process algorithms, one synchronous (CBJ) and one asynchronous (ABT), and one multiple-process algorithm - concurrent dynamic backtracking ($ConcDB$). The performance of the three representative algorithms is evaluated in section 6 and the impact of delayed messages on their performance is presented. The impact of delayed messages on each of the algorithms is found to be related to the properties of the algorithm's family and will be explained in the discussion (Section 7).

5.1 Conflict Based Backjumping

The Synchronous Backtrack algorithm (SBT) [18], is a distributed version of chronological backtrack [11]. 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 variable 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

– **CBJ**:
1.done ← false
2.**if**(first_agent)
3. $CPA \leftarrow$ create_CPA
4. assign_CPA
5. **while**(**not** done)
6. **switch** msg.type
7. $stop$:done ← true
8. $backtrack$: remove_last_assignment
9. assign_CPA
10. CPA:refresh_domain
11. assign_CPA

– **assign_CPA**:
1. CPA ← assign_local
2. **if**(is_consistent(CPA))
3. **if**(is_full(CPA))
4. $report\_solution$
5. stop
6. **else**
7. send(CPA, next)
8. **else**
9. $backtrack$

– **backtrack**:
1. CPA ← $resolve\_nogoods$
2. **if**(is_empty(CPA))
3. CPA ← no_solution
4. stop
5. **else**
6. send($backtrack$, $CPA.last\_asignee$)

– **remove_last_assignment**:
1. store(CPA.last_assignment,CPA)
2. CPA ← $\{last\_sent\_CPA\} \setminus \{last\_assignment\}$
3. current_domain ← $\{current\_domain\} \setminus \{last\_assignment\}$

– **refresh_domain**:
1. **for each** stored $Nogood\ sn$
2. **if**(not consistent(CPA,sn))
3. current_domain ← $\{current\_domain\} \cup \{sn.last\_assignment\}$

– **stop**:
1. send(stop, all_other_agents)
2. done ← true

Fig. 2. The Distributed CBJ algorithm

last agent manages to find a consistent assignment for its variable. The algorithm ends unsuccessfully if the first agent encounters an empty domain [18].

The synchronous (distributed) version of Conflict Based Backjumping (CBJ) improves on simple synchronous backtrack (SBT) by using a method based on dynamic backtracking [5]. In the improved version, 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. 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) [5]. 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.

The CBJ algorithm is presented in figure 2. In the main function, the first agent initializes the search by creating a CPA, assigning and sending it by using the function $assign\_CPA$ (lines 2 - 4). Lines 5 to 10 describe how agents respond to one of three types of messages:

1. $stop$: indicating that the search has ended.
2. CPA: carrying a CPA forward.
3. $backtrack$: carrying a CPA backwards, with an inconsistent assignment.

Upon the reception of a stop message the agent simply stops the search by exiting the loop. When a CPA moving forward is received, the agent first calls function $refresh\_domain$. This returns to the agent's $current\_domain$ values whose explanation is not included in the received CPA. Next, the agent calls function $assign\_CPA$, attempting to assign its variable.

When a $backtrack$ message is received, the agent calls function $remove\_last\_assignment$ which removes the value assignment of the agent in the inconsistent CPA from its $current\_domain$. It then stores it with the received CPA in the form of a $Nogood$. Finally, it replaces the CPA with a copy of the last CPA it sent, which holds the assignment it will try to extend and send forward. This takes place in the function $assign\_CPA$ that is called immediately after $remove\_last\_assignment$. When the agent fails to extend a CPA it calls function *backtrack* whose first line resolves the inconsistent subset of the CPA (line 1). Then, a check is made whether the $Nogood$ created is empty which will indicate the $DisCSP$ has no solution (lines 2-4). If the $Nogood$ found is not empty, it is sent to the agent with the lowest priority whose assignment is included in the $Nogood$ (lines 6). This is standard dynamic backtracking [5].

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

The ABT algorithm [18], 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 [18]), it updates

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

– **when received** (**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. 3. The ABT algorithm with full $Nogood$ recording

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 [18]). 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, 18].

The performance of ABT can be strongly improved by requiring agents to read all messages they receive before performing computation [18]. 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 [21]. 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 [21] 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 3 [18]. 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 $check\_agent\_view$.

5.3 Concurrent Dynamic Backtracking

In order to present concurrent dynamic backtracking a simpler more general version, *ConcBT* is first presented, followed by the changes in order to add dynamic backtracking to the algorithm. The $ConcBT$ algorithm [22] 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 consistent 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 [22].

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 manner (see [22] 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. In the description of the $ConcBT$ algorithm the following terminology is used:

- $CPA\_generator$: Every *CPA* carries the *ID* of the agent that created it.
- $steps\_limit$: the number of steps (from one agent to the next) that will trigger a split, if the *CPA* does not find a solution, or return to its generator.
- $split\_set$: the set of SP-IDs, stored in each SP, including the IDs of the active SPs that were split from the SP by the agent holding the $split\_set$.
- $origin\_SP$: an agent that performs a dynamic split, holds in each of the new *SP*s the *ID* of the *SP* it was split from (i.e. of $origin\_SP$). An analogous definition holds for $origin\_CPA$. The $origin\_SP$ of an *SP* that was not created in a dynamic split operation is its own *ID*.

The messages exchanged by agents in $ConcBT$ are:

- **CPA** - a regular CPA message.
- **backtrack** - a CPA sent in a backtrack operation.
- **stop** - a message indicating the end of the search.
- **split** - a message that is sent in order to trigger a split operation. Contains the *ID* of the *SP* to be split.

The $ConcBT$ algorithm is presented in Figure 4 and the detailed description of its different functions is as follows.

- The main function **ConcBT**, initializes the search if it is run by the *initializing agent (IA)*. It initializes the algorithm by creating multiple *SP*s, assigning each SP

– **ConcBT**:
1. done ← false
2. **if**(IA) **then** $initialize\_SPs$
3. **while**(**not** done)
4. **switch** msg.type
5. $split$: perform_split
6. $stop$: done ← true
7. CPA: receive_CPA
8. $backtrack$: receive_CPA

– **initialize_SPs**:
1. **for** i ← 1 to $domain\_size$
2. create_SP(i)
3. SP[i].domain ← domain.val[i]
4. CPA ← create_CPA(i)
5. $assign\_CPA$

– **receive_CPA**:
1. CPA ← $msg.CPA$
2. **if**(first_received(CPA_ID))
3. create_SP(CPA_ID)
4. **if**($CPA\_generator$ = ID)
5. CPA_steps ← 0
6. **else**
7. CPA_steps ++
8. **if**(CPA_steps = $steps\_limit$)
9. send($split$, $CPA\_generator$)
10. **if**(msg.type = $backtrack$)
11. remove_last_assignment
12. $assign\_CPA$

– **assign_CPA**:
1. CPA ← assign_local
2. **if**(is_consistent(CPA))
3. **if**(is_full(CPA))
4. $report\_solution$
5. stop
6. **else**
7. send(CPA, next_agent)
8. **else**
9. $backtrack$

– **backtrack**:
1. delete(current_CPA from $origin\_split\_set$)
2. **if**($origin\_split\_set$ is_empty)
3. **if**(IA)
4. CPA ← no_solution
5. **if**(no_active_CPAs)
6. report_no_solution
7. stop
8. **else**
9. send($backtrack$, $last\_assignee$)
10. **else**
11. $mark\_fail(current\_CPA)$

– **perform_split**:
1. **if**(**not_backtracked**(CPA))
2. var ← $select\_split\_var$
3. **if**(var is_not null)
4. create_split_SP(var)
5. create_split_CPA(SP_ID)
6. add(SP_ID to $origin\_split\_set$)
7. $assign\_CPA$
8. **else**
9. send($split$, $next\_agent$)

– **stop**:
1. send(stop, all_other_agents)
2. done ← true

Fig. 4. The ConcBT algorithm

with one of the first variable's values. After initialization, it loops forever, waiting for messages to arrive.

– **receive_CPA** first checks if the agent holds a *SP* with the ID of the *current_CPA* and if not, creates a new *SP* (lines 2,3). If the CPA is received by its generator, it changes the value of the steps counter (CPA_steps) to zero (lines 4,5). This prevents unnecessary splitting. Otherwise, it checks whether the *CPA* has reached the $steps\_limit$ and a split must be initialized (lines 7-9). Before assigning the CPA a check is made whether the CPA was received in a $backtrack$, if so the

previous assignment of the agent which is the last assignment made on the CPA is removed, before $assign\_CPA$ is called (lines 10-11).

- **assign_CPA** tries to find an assignment for the local variables of the agent, which is consistent with the assignments on the $current\_CPA$. If it succeeds, the agent sends the *CPA* to the selected $next\_agent$ (line 7). If not, it calls the *backtrack* method (line 9).
- The **backtrack** method is called when a consistent assignment cannot be found in a *SP*. Since a split might have been performed by the current agent, a check is made, whether all the *CPA*s that were split from the $current\_CPA$ have also failed (line 2). When all split CPAs have returned unsuccessfully, a backtrack message is sent carrying the ID of the $origin\_CPA$. In case of an *IA*, the *origin_SP* is marked as a failure (lines 3-4). If all other *SP*s are marked as failures, the search is ended unsuccessfully (line 6).
- The **perform_split** method tries to find in the *SP* specified in the $split\_message$, a variable with a non-empty current_domain. It first checks that the CPA to be split has not been sent back already, in a backtrack message (line 1). If it does not find a variable for splitting, it sends a split_message to $next\_agent$ (lines 8-9). If it finds a variable to split, it creates a new *SP* and CPA, and calls *assign_CPA* to initialize the new search (lines 3-5). The *ID* of the generated CPA is added to the split set of the divided SP's $origin\_SP$ (line 6).

The best version of concurrent backtracking search uses methods of backjumping that are based on *Dynamic Backtracking* [5]. Each agent that removes a value from its current domain stores the partial assignment that caused the removal of the value. This stored partial assignment is called an *eliminating explanation* by [5]. When the current domain of an agent empties, the agent constructs a backtrack message from the union of all assignments in its stored removal explanations. The union of all removal explanations is an inconsistent partial assignment, or a *Nogood* [5, 18]. The backtrack message is sent to the agent which is the owner of the most recently assigned variable in the inconsistent partial assignment. This version of concurrent search is called *Concurrent Dynamic Backtracking (ConcDB)*.

In concurrent dynamic backtracking, a short *Nogood* can rule out multiple sub-search-spaces, all of which contain no solution and are thus unsolvable. In order to terminate the corresponding search processes, an agent that receives a backtrack message performs the following procedure:

- Detect the SP to which the received (backtrack) CPA either belongs or was split from.
- Check if the CPA corresponding to the detected SP was split down its path.
- If it was:
  - Send an $unsolvable$ message to the *next_agent* of the related SP, thus generating a series of messages along the path of the CPA.
  - choose a new unique ID for the CPA received and its related SP.
  - continue the search using the SP and CPA with the new ID.
- Check if there are other SPs which contain the received inconsistent partial assignment (by calling function ***check_SPs***). Send corresponding $unsolvable$ messages and resume the search on them with new generated $CPAs$.

ConcDB:
..
..
..
9. $unsolvable$: mark_unsolvable(msg.SP)

receive_CPA:
1. CPA $\leftarrow msg.CPA$
2. **if**(unsolvable SP)
3. terminate CPA
4. else
..
..
..
14. **if**(msg.type = $backtrack$)
15. check_SPs(CPA.inconsistent_assignment)
16. last_sent_CPA.$remove\_last\_assignment$
17. $CPA \leftarrow last\_sent\_CPA$
18. **if**(SP_split_ahead)
19. send(unsolvable, SP.next_agent)
20. rename_SP
21. $assign\_CPA$

backtrack:
..
..
9. $backtrack\_msg \leftarrow$ $inconsistent\_assignment$
10. send($backtrack\_msg$, $lowest\_priority\_assignee$)
11. **else**
12. $mark\_fail(current\_CPA)$

mark_unsolvable(SP)
1. mark SP unsolvable
2. send(unsolvable, SP.next_agent)
3. **for each** split_SP in SP.origin.split_set
4. mark split_SP unsolvable
5. send(unsolvable, split_SP.next_agent)

check_SPs(inconsistent_assignment)
1. **for each** sp in $\{SPs \setminus current\_SP\}$
2. **if**(sp.contains(inconsistent_assignment))
3. send(unsolvable, sp.next_agent)
4. last_sent_CPA.$remove\_last\_assignment$
5. $CPA \leftarrow$ last_sent_CPA
6. sp.rename_SP
7. $assign\_CPA$

Fig. 5. Methods for Dynamic Backtracking of $ConcDB$

The *unsolvable* message used by the $ConcDB$ algorithm, is a message not used in *Concurrent Backtracking (ConcBT)*, which indicates an unsolvable sub-search-space. An agent that receives an $unsolvable$ message performs the following operations for the unsolvable SP and each of the SPs split from it:

- mark the SP as unsolvable.
- send an $unsolvable$ message which carries the ID of the SP to the agent to whom the related CPA was last sent.

The change of ID makes the resumed search process independent of the process of terminating unsolvable search spaces. If the agents would have resumed the search using the ID of the original SP or of the received CPA, a race condition would arise since there is no synchronization between the process of terminating unsolvable search procedures to the resumed valid search procedure. In such a case, an agent that received an *unsolvable message* might have marked an active search space as unsolvable.

Figure 5 presents the methods ***ConcDB***, ***receive_CPA*** and ***backtrack***, that were changed from the general description of *Concurrent Search* in Figure 4, followed by two additional methods needed for adding *Dynamic Backtracking* to concurrent search. Let us look at the differences in code. In method **receive_CPA** a check is made in lines 3,4 whether the SP related to the received CPA is marked unsolvable. In such a case

the CPA is not assigned and the related SP is terminated. For a backtracking CPA (lines 14-20) a check is made whether there are other SPs which can be declared unsolvable. This can happen when the head (or prefix) of their partial assignment (their *common head i.e. CH*) contains the received inconsistent partial assignment. For such a case, procedure **check_SPs** is called, which for every such SP found, initiates the termination of the search process on the unsolvable sub-search-space and resumes the search with a newly generated CPA. Then a check is made whether the SP was split by agents who received the CPA after this agent (line 18). If so, the termination of the unsolvable SP is initiated by sending an $unsolvable$ message. A new ID is assigned to the received CPA and to its related SP (line 20).

In method $backtrack$, the agent inserts the culprit inconsistent partial assignment into the backtrack message (line 9) before sending it back in line 10. This is the only difference from the standard backtrack method in Figure 4.

As described above, method **mark_unsolvable** is part of the mechanism for terminating SPs on unsolvable search spaces. The agent marks the SP related to the message received, and any SP split from it, as unsolvable and sends unsolvable messages to the agents to whom the corresponding CPAs were sent.

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. [12, 16]). 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 ($NCCCs$) [9, 23], to measure computational cost. This measures the combined path of computation, from beginning to end, in terms of constrait checks. The other measure used is the communication load, in the form of the total number of messages sent [8]. 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 three algorithms are compared without any message delay. The results presented in Figures 6 and 8 show that the numbers of non-concurrent constrait checks that the three algorithms perform are very similar, on systems with no message delays. ABT performs slightly less steps than CBJ and $ConcDB$ performs slightly better than ABT. When it comes to network load, the results in figures 7 and 9 show that for the harder problem instances, agents in ABT send between

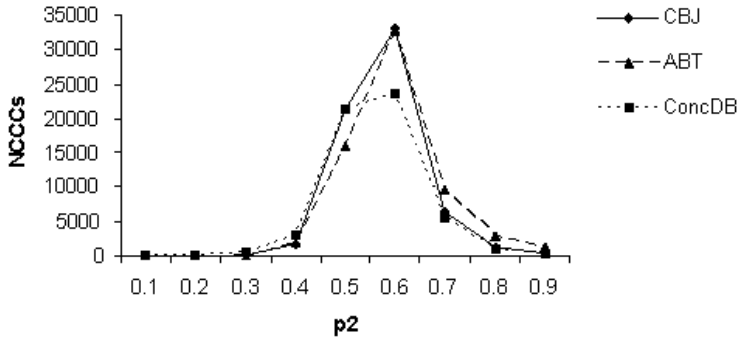

Fig. 6. Non-concurrent constraint checks with no message delays ($p_1 = 0.4$)

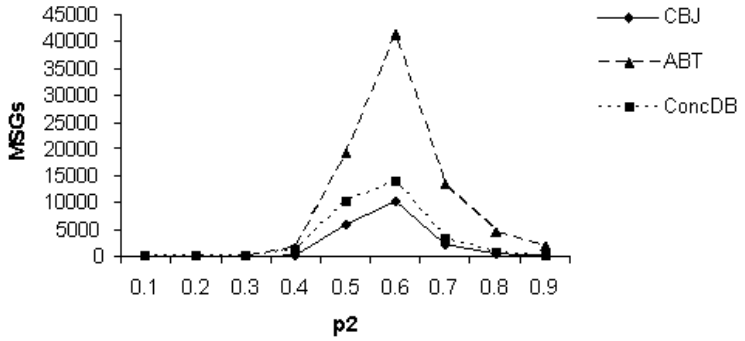

Fig. 7. Total number of messages with no message delays ($p_1 = 0.4$)

4 to 5 times *more messages* than sent by agents running CBJ and $ConcDB$. All four figures: 6, 7, 8 and 9 show clearly the presence of a phase transition ([19]).

In the second set of experiments, messages were delayed randomly for 50-100 non-concurrent constraint checks (as described in section 3). Figure 10 presents the number of non concurrent constraint checks performed by the three algorithms running on sparse DisCSPs with random message delays. The most obvious result of Figure 10 is that CBJ is affected most from message delay. This could have been expected. Since CBJ performs no concurrent computation the total amount of message delay is added to the runtime of the algorithm. This gives a large effect on the run-time results. Figure 11 presents a closer look at the results of ABT and $ConcDB$ in this run. While $ConcDB$ performed about 40% more $NCCCs$ than the number of $NCCCS$ it per-

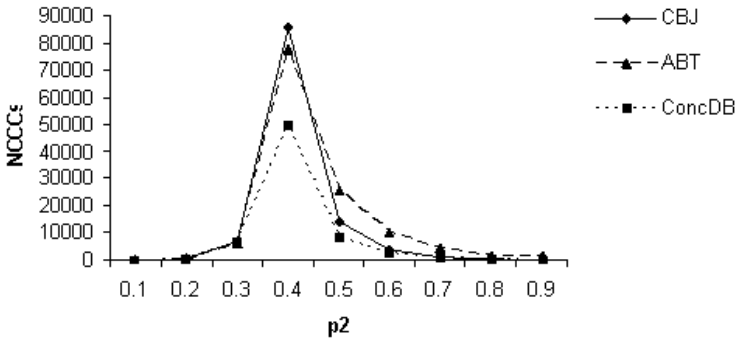

Fig. 8. Non-concurrent constraint checks with no message delays ($p_1 = 0.7$)

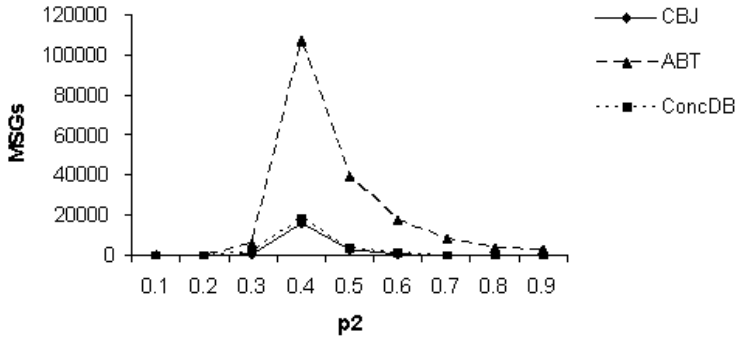

Fig. 9. Total number of messages with no message delays ($p_1 = 0.7$)

formed with no message delays, ABT performes *three times more* $NCCCs$ than it does for perfect communication.

Figure 12 presents the total number of messages sent by the three algorithms with random message delays. It is interesting to see that while the total number of messages sent by CBJ and $ConcDB$ are not affected by message delay, the number of messages sent by ABT grows by a factor of 2.

Figures 13, 14 and 15 show similar results for denser DisCSPs ($p_1 = 0.7$).

The third set of experiments investigates the impact of the size and range of the random delays on the different algorithms. The effect of varying the delay size on a sequential assignment (synchronous) single search algorithm is easy to understand. In order to investigate the behavior of algorithms which perform concurrent computation, in the

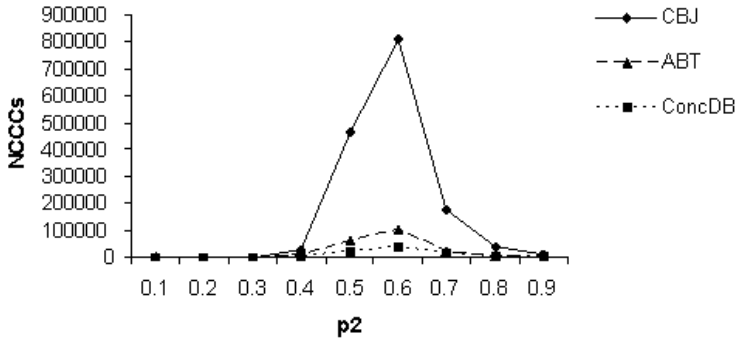

Fig. 10. Non-concurrent constraint checks with random message delays ($p_1 = 0.4$)

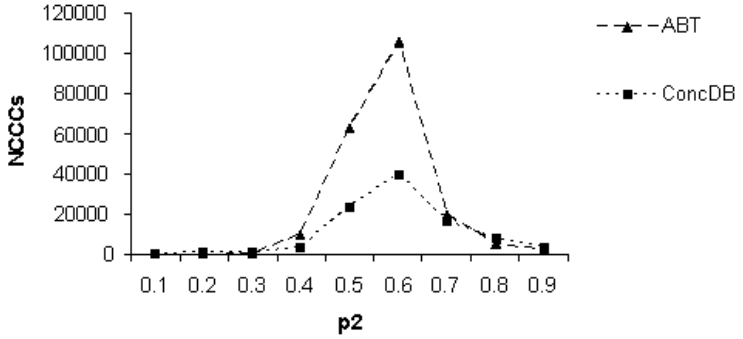

Fig. 11. A closer look at NCCCs performed by ABT and ConcDB, with random message delays ($p_1 = 0.4$)

presence of message delays of different sizes and range, experiments were performed for the harder problem instances. The algorithms were run with an increasing amount and range of message delay, on the hardest problem instances (tightness $p_2 = 0.6$ for $p_1 = 0.4$ and $p_2 = 0.4$ for $p_1 = 0.7$). The impact of random delays on the different algorithms is presented in Figures 16 and 17. The number of non-concurrent constraint checks of the single search algorithm (ABT) grows with the size of message delay. In contrast, larger delays have a small impact on the number of non-concurrent constraint checks performed by concurrent search ($ConcDB$).

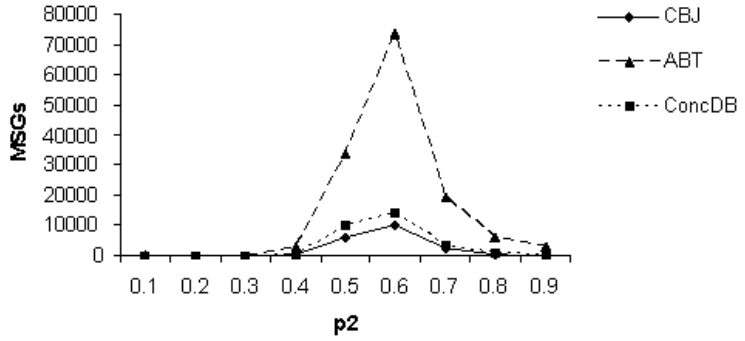

Fig. 12. Total number of messages with random message delays ($p_1 = 0.4$)

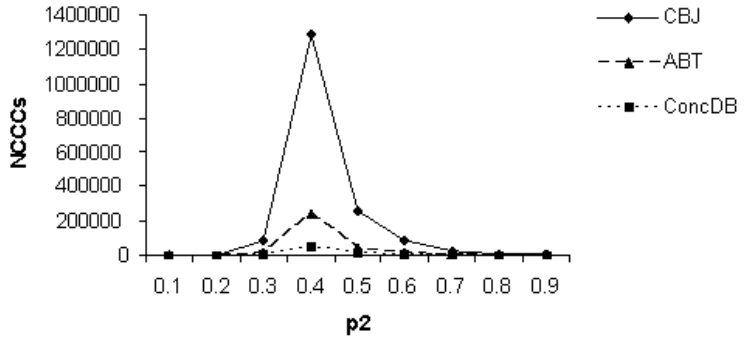

Fig. 13. Non-concurrent constraint checks with random message delays ($p_1 = 0.7$)

7 Discussion

Three sets of experiments to investigate the effect of message delays on the performance of $DisCSP$ search algorithms were performed. Distributed search algorithms on $DisCSP$s fall into two distinct families - Single search process algorithms and concurrent search algorithms. The experiments compared three representative search algorithms of these two families - Synchronous BT; Asynchronous BT; and Concurrent DB.

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

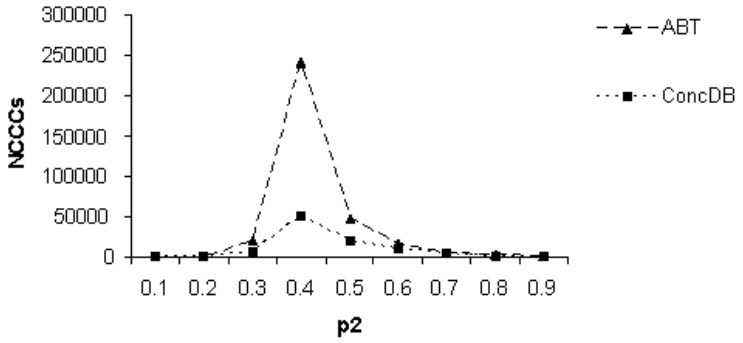

Fig. 14. A closer look on the NCCCs performed by of ABT and ConcDB, with random message delays ($p_1 = 0.7$)

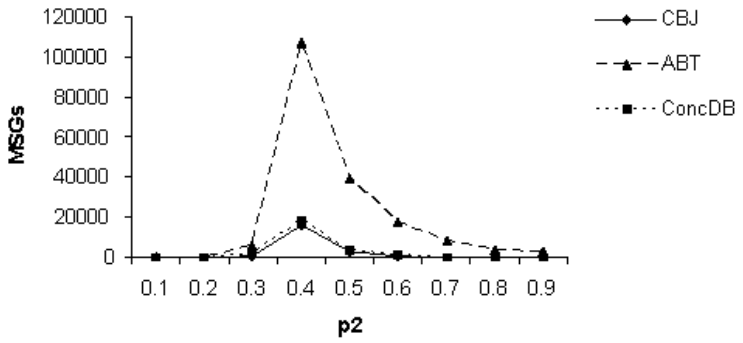

Fig. 15. Total number of messages with random message delays ($p_1 = 0.7$)

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 into account. When the number of non-concurrent CCs is calculated, it reveals a large impact of message delay on the performance of single process algorithms. In other words, the actual time it would take CBJ to report a solution (including the delays of message) is much longer than that of $ConcDB$ or ABT.

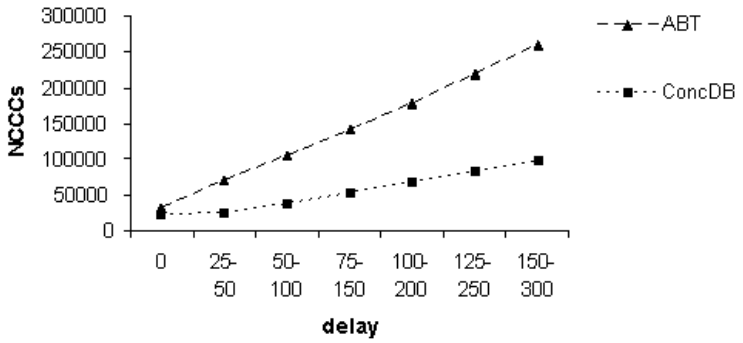

Fig. 16. Number of non-concurrent CCs vs. size of random message delays ($p_1 = 0.4$)

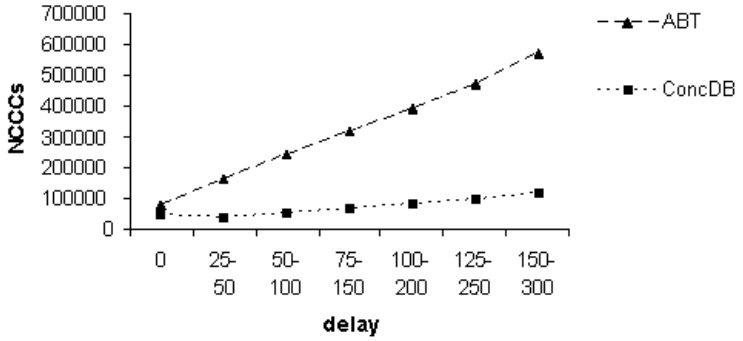

Fig. 17. Number of non-concurrent CCs vs. size of random message delays ($p_1 = 0.7$)

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 (cf. [1, 15]). The impact is not as strong as in synchronous CBJ (Figures 10, 12, 13 and 15).

In order to further investigate the behaviour of the algorithms in the presence of message delay the simple method for counting non-concurrent constraint checks of [9] (see Section 3) can be performed concurrently during the run of the $AMDS$ simulator. This would give us the number of $NCCCs$ which were actualy performed without the addition of messge delays to the final result. Figures 18 and 19 present the actual count of non-concurrent constraints checks (without adding delays) performed by the

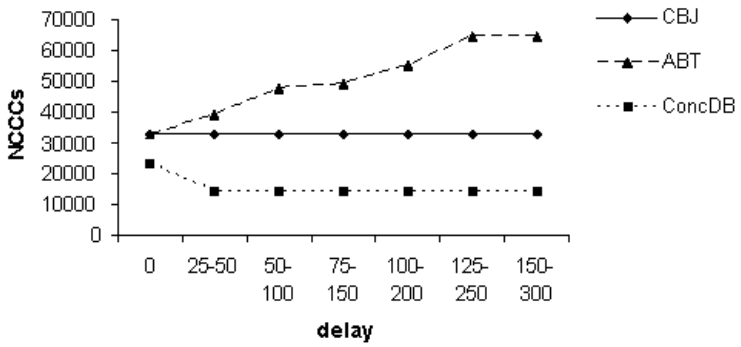

Fig. 18. Number of non-concurrent CCs actualy performed vs. size of random message delays ($p_1 = 0.4$)

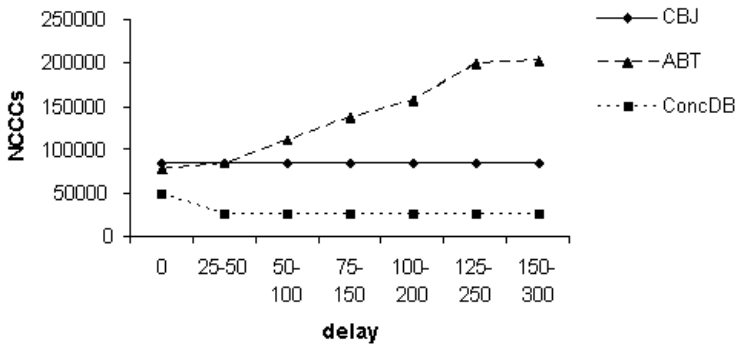

Fig. 19. Number of non-concurrent CCs actualy performed vs. size of random message delays ($p_1 = 0.7$)

agents during the algorithm run. As expected, CBJ performs exactly the same number of $NCCCs$ as with no delays. The number of NCCCs performed by ABT in the presence of delays, *grows by a factor of 2*. This illuminates an important feature of the standard simulation of runs of $DisCSP$ algorithms. Based on instantaneous arrival of messages, ABT reads multiple messages at each step. With random message delays, 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 conflicting. As a result, agents perform more unnecessary computation steps when responding to fewer messages in each cycle. The improvement that results

from reading all incoming messages in each step [18], is no longer useful when messages have random delays. This can explain a similar result for ABT, on a different set of problems [1]. As can be seen in Figures 18 and 19, for a multiple search process algorithm, like $ConcDB$, the number of actual non-concurrent CCs is not affected and even decreases by the delay of messages.

To illuminate the robustness of $ConcDB$ to message delay imagine the following example. Consider the case where $ConcDB$ 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, this potential waiting is caused by the system. By choosing the right $split\_limit$, agents can be kept busy at all times, performing computation against consistent partial assignments. The results in section 6 demonstrate that the above claim can be achieved.

In terms of network load, the results of section 6 show that asynchronous backtracking puts a heavy load on the network, which doubles in the case of message delays. The number of messages sent in concurrent search algorithms, is always much smaller and is affected very lightly by message delays.

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. Two families of $DisCSP$ search algorithms have been presented and investigated. Single process algorithms (SPAs) and multiple process algorithms (MPAs or concurrent search). The results imply that, single process algorithms (SPAs), are much more affected by message delays, than concurrent search. The number of NCCCs grows linearly with message delay for completely synchronous algorithms like CBJ. The impact on asynchronous backtracking, (ABT), is large. Both the computational effort and the load on the network grow by a large factor, although the effect on runtime is smaller than that of CBJ. This strengthens the results of [15, 1].

The concurrent search algorithm $ConcDB$ shows the highest robustness to message delays. This is connected to the fact that in $ConcDB$ agents always perform computation against consistent partial assignments. Computation performed in one sub-search-space while others are delayed is not wasted as in asynchronous backtracking. The effect of message delay on concurrent search is minor in terms of non concurrent constraint checks as well as on its network load.

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] I. Brito and P. Meseguer. Synchronous,asnchronous and hybrid algorithms for discsp. In *Workshop on Distributed Constraints Reasoning(DCR-04) CP-2004*, Toronto, September 2004.

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

[5] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[6] Y. Hamadi. Interleaved backtracking in distributed constraint networks. *Intern. Jou. AI Tools*, 11:167–188, 2002.

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

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

[9] 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.

[10] A. Meisels and R. Zivan. Asynchronous forward-checking for distributed csps. In W. Zhang, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[11] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

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

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

[14] M. C. Silaghi and B. Faltings. Parallel proposals in asynchronous search. Technical Report 01/#371, EPFL, August 2001. http://liawww.epfl.ch/cgi-bin/Pubs/recherche.

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

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

[17] 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.

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

[19] 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.

[20] R. Zivan and A. Meisels. Parallel backtrack search on discsps. In *Proc. Workshop on Distributed Constraint Reasoning DCR-02*, Bologna, July 2002.

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

[22] R. Zivan and A. Meisels. Concurrent backtrack search for discsps. In *Proc. FLAIRS-04*, pages 776–81, Maiami Florida, May 2004.

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