CP04 Tutorial: Distributed Constraints Satisfaction Algorithms, Performance, Communication [*]

Amnon Meisels
am@cs.bgu.ac.il

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

Abstract. Distributed constraints satisfaction problems (DisCSPs) have been studied for over a decade. The first distributed search algorithm was asynchronous backtracking, which is still the most studied. In the last few years, several new families of distributed search algorithms have been investigated and comparative experimental studies are encouraging.
A natural extension to distributed constraints satisfaction is distributed constraints optimization. Stochastic search algorithms for solving $DisCSPs$, such as Distributed Breakout, have appeared a few years ago. Distributed stochastic search algorithms are naturally suitable for solving distributed optimization. In contrast, asynchronous search algorithms for distributed optimization have been proposed in recent years.
Due to the distributed nature of the problem, message delay can have unexpected effects on the behavior of algorithms on $DisCSPs$. This has been shown in an experimental study that induced random delays on messages sent among agents. In order to study the impact of message delays on DisCSP search, a model of delays in terms of concurrent performance measures is needed. Within such a model, the behavior of families of search algorithms in the presence of delays is varied and interesting.
An important feature of the distribution of the problem among agents is their ability to maintain some privacy. Agents may not want to share their values with other agents, and they may wish to keep constraints as private as possible. Some recent work has resulted in versions of asynchronous backtracking that maintain both privacy of values and privacy of constraints. Other investigations of privacy in $DisCSPs$ focused on the analysis of information gain by studying a well-defined problem, that of scheduling meetings of agents.

1 Introduction

Distributed constraint 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. [29, 27, 24]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages with other agents, to check consistency

[*] Partially supported by the Lynn and William Frankel Center for Computer Science

of their proposed assignments against constraints with variables owned by different agents [4].

Distributed CSPs are an elegant model for many every day combinatorial problems that are distributed by nature. Take for example a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint satisfaction problem for each ward. Some of the nurses in every ward are qualified to work in the *Emergency Room*. Hospital regulations require a certain number of qualified nurses (e.g. for Emergency Room) in each shift. This imposes constraints among the timetables of different wards and generates a complex Distributed CSP [24].

An example of a large scale DisCSP has started as a DARPA problem presented publicly on the web in 2000. The description here is from [2]. The problem has n sensors and m targets, where a target is tracked if k sensors are tracking it at the same time. The major constraint is that a sensor can only track one target at a time. Bejar et. al. have formulated this problem as a DisCSP as follows. Each sensor is represented by an agent. Each agent has variables for every target that is in range. Variables are assigned the value 1 if the agent selects to track them and 0 otherwise. Each agent is constrained internally to have only one variable with the value 1. Constraints between agents are such that for every target, there are at least k agents that have assigned 1 to the value of their variable that corresponds to that target. Bejar et. al. have termed this DisCSP problem SensorCSP [2].

SensorCSP can be a large problem and has attracted researchers to investigate non complete DisCSP search algorithms on them [34, 20]. The most popular distributed stochastic search algorithm has been the Distributed Breakout Algorithm (DBA) [28, 31]. A comparative study of distributed stochastic search looked at DBA and at distributed local search [31]. The main problem with the family of distributed stochastic search algorithms is their synchronous behavior (cf. [32]. All agents running the Distributed Breakout Algorithm, for example, need to select a value (or replace weights) at synchronous steps [32]. SensorCSPs were also used to investigate the experimental behavior of distributed search algorithms under communication delays [6]. The impact of communication delays on asynchronous backtracking algorithms was found to be quite strong and performance was found to deteriorate strongly with larger random delays [6].

A search procedure for a consistent assignment of all agents in a distributed CSP ($DisCSP$), is a distributed algorithm. All agents cooperate in search for a globally consistent solution. The solution involves assignments of all agents to all their variables and exchange of information among all agents, to check the consistency of assignments with constraints among agents. An intuitive way to make the distributed search process on DisCSPs efficient is to enable agents to compute concurrently. Concurrent computation by agents can result in a shorter overall time of computation for finding a solution. Section 2 presents the state of the art of distributed search algorithms on DisCSPs. Incomplete search algorithms that solve distributed optimization problems on $DisCSPs$ are presented too.

Search algorithms on DisCSPs must be measured in terms of distributed computation. Two measures are commonly used to evaluate distributed algorithms - time, which is measured in terms of computational effort, and network load [11]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [27]). 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 ([13, 22]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [11]. A sample of performance measurements of complete search algorithms on $DisCSPs$ are presented in section 5.

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 [27]. 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 [27]. 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. The impact of message delays on DisCSP algorithms is discussed in section 3 and some interesting experimental results are given in section 5. It turns out that the performance of asynchronous backtracking and asynchronous forward checking algorithms deteriorates strongly with large enough random message delays. In contrast, concurrent search algorithms are quite robust to random message delays [37]. This is probably connected to the fact that the multiple search processes compensate one another in the presence of random message delays (see section 3).

An important goal of search algorithms for the distributed constraint satisfaction problem is to support the privacy of agents. During cooperative search for a globaly consistent solution, agents exchange messages about their assignments and about conflicts with other agents' assignments. This creates a natural trade-off between information disclosure and the efficiency (and correctness) of the distributed search process. The first to investigate measures of privacy for $DisCSPs$ were Meseguer et. al. [**?**]. In a series of two papers they presented algorithms for maintaining two types of privacy during the run of the asynchronous backtracking (ABT) algorithm [15, **?**]. Privacy of assignments and privacy of constraints are described in section 4.

A different approach for investigating the privacy of distributed search was presented first by Wallace and Freuder [26]. They use a concrete family of problems (scheduling meetings among agents) to compare the amount of needed computations for finding a solution, when different quantities of information were exchanged among the searching agents [25]. In section 4 the meeting scheduling problem of [26] is presented and generalized. Some new results about the trade-off between volunteered information and the efficiency of search are presented.

2 Search algorithms on DisCSPs

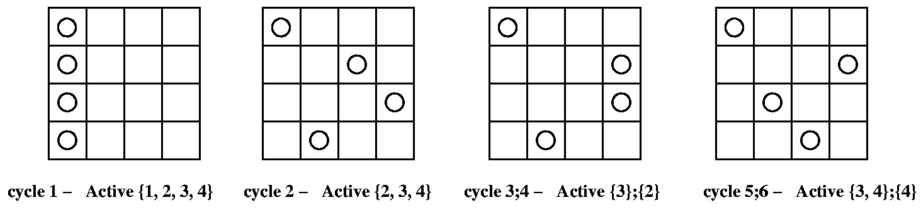
One method for achieving concurrency in search on Distributed CSPs is to enable agents to cooperate in a single backtrack procedure. In order to avoid the waiting time of a single backtrack search, agents compute assignments to their variables asynchronously. In asynchronous backtracking algorithms, agents assign their variables without waiting to receive information about all relevant assignments of other agents [29, 22]. In order to make asynchronous backtracking correct and complete, all agents share a static order of variables and the algorithm keeps data structures for $Nogoods$ that are discovered during search (cf. [4]).

Asynchronous backtracking (ABT) uses a complete order among all agents. Agents receive messages informing them about assignments of agents that are ahead of them in the total order (i.e. **ok?** messages). After performing an assignment, each agent sends **ok?** messages to agents that are ordered after it. In ABT, agents always have their variables assigned. Initially, variables are assigned without waiting for messages informing about assignments of other (constraining) agents. When messages informing of assignments of other agents, that are conflicting with the current assignment of the receiving agent arrive, the receiving agent performs one of two actions. Either it finds an alternative assignment that is consistent with the received message. Or, it sends back a $Nogood$ message. The $Nogood$ message informs the receiving agent that its assignment has to be changed. Having sent back this $Nogood$ (backtracking) message, the agent than assumes that the culprit assignment will be changed and proceeds to assign its variables accordingly. This is the way in which all agents running ABT keep being assigned at all times (cf. [27]).

The performance of ABT can be strongly improved by requiring agents to read all messages they receive before performing computation [28]. 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 status it keeps of all other agents' assignments (i.e. its $Agent\_view$) 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 [36]. 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 sections 3 and 5).

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 [7]. A version of ABT that uses this method was presented in [4]. In [36] 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 quoted in this tutorial 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.

In order to gain some intuitive understanding of ABT, consider its run on a simple problem. Figures 1, 2 describe the 10 cycles of ABT, solving the 4-queens problem. Each cycle of computation includes the receiving of messages, computations triggered by the received messages, and the sending of messages [28, 27]. The four agents A_1, A_2, A_3, A_4 are ordered from top to bottom. In the first cycle, all agents assign the first value in their domains (i.e. the first column) and send **ok?** messages downstream. In the second cycle Agents A_1, A_2, A_3 recieve the **ok?** messages sent to them and proceed to assign consistent values to their variables. Agent A_3 assigns the value 3 that is consistent with the assignments of A_1 and A_2 that it received. Agent A_4 has no consistent value with the assignments of A_1, A_2, A_3. It sends a $Nogood$ containing these three assignments to A_3 and removes the assignment of A_3 from its $Agent\_View$. Then, it assigns the value 2 that is consistent with the assignments that it recieved from A_1, A_2. The active agents in this cycle are A_2, A_3, A_4.



cycle 1 – Active {1, 2, 3, 4}
cycle 2 – Active {2, 3, 4}
cycle 3;4 – Active {3};{2}
cycle 5;6 – Active {3, 4};{4}

Fig. 1. First 6 cycles of ABT - lists of Active Agents in each cycle

In the third cycle, for example, only agent A_3 is active. After receiving the assignment of agent A_2, it sends back a $Nogood$ message to agent A_2. It then erases the assignment of agent A_2 from its $Agent\_View$ and validates that its current assignment (the value 4) is consistent with the assignments of agent A_1. Agents A_1 and A_2 continue to be idle, having received no messages that were sent in cycle 2. The same is true for agent A_4. Agent A_3 receives also the $Nogood$ sent by A_4 in cycle 2, but, ignores it since it includes an invalid assignment of A_2 (i.e. $< 2, 1 >$ and not the currently correct $< 2, 4 >$).

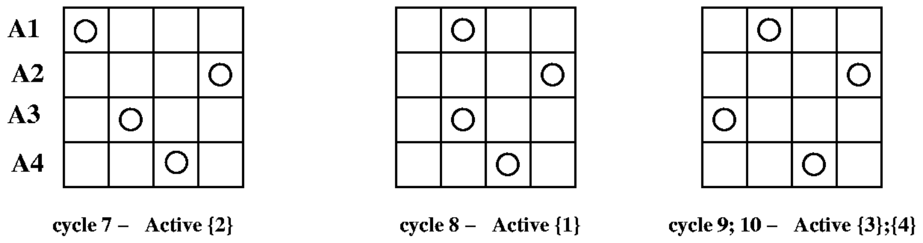


Fig. 2. Cycles 7-10 of ABT - lists of Active Agents in each cycle

To take another example, consider cycle 6. Agent A_4 receives the new assignment of agent A_3 and sends it a $Nogood$ message. Having erased the assignment of A_3 after sending the $Nogood$ message, it then decides to stay at its current assignment (the value 3), since it is compatible with agents A_1 and A_2. Agent A_3 is idle in cycle 6, since it receives no messages from either agent A_1 or A_2 (who are idle too). So, A_4 is the only active agent at cycle 6. After cycle 10 all agents remain idle, having no constraint violations with assignments on their $Agent\_Views$ [28].

A different family of distributed search algorithms on *DisCSP*s, is termed *Asynchronous Forward-Checking (AFC)*. The *AFC* algorithm processes forward checking (FC) asynchronously [14]. In the *AFC* algorithm, the state of the search process is represented by a data structure called *Current Partial Assignment (CPA)*. The *CPA* starts empty at some initializing agent that records its assignments on it and sends it to the next agent. Each receiving agent adds its assignment to the *CPA*, if a consistent assignment can be found. Otherwise, it backtracks by sending the same *CPA* to a former agent to revise its assignment on the *CPA*.

Each agent that performs an assignment on the *CPA* sends forward a copy of the updated *CPA*, requesting all agents to perform forward-checking. Agents that receive copies of assignments filter their domains and in case of a dead-end send back a *Not_OK* message. The concurrency of the *AFC* algorithm is achieved by the fact that forward-checking is performed concurrently by all agents. The protocol of the *AFC* algorithm enables agents to process forward checking (FC) messages concurrently and yet block the assignment process at the agent that violates consistency with future variables. On hard instances of randomly generated *DisCSP*s with different message delays, *AFC* outperforms ABT by a large factor [14] (see section 5).

Recently, an interesting improvement to *AFC* was proposed. In addition to concurrency of checking forward, a concurrency of backtracking was introduced [19]. When a backtrack is initiated by a *Not_OK* message, it is sent directly to the culprit agent. This triggers an additional search process, starting at the backtracking agent. An intuitive way to understand this improvement to *AFC* is to say that it adds concurrent backtracking processes to its asynchronous forward-checking. All the generated concurrent search processes, save one, are unsolvable (i.e. contain a $Nogood$ that generated the backtrack message). Consequently, the improved *AFC* algorithm terminates all of these search processes as soon as their unsolvability is validated (within a small number of steps). The initial experimental report for the improved *AFC* algorithm shows an encouraging improvement of performance [19].

A different way of achieving concurrency for search on *DisCSPs*, both from asynchronous backtracking and from asynchronous forward-checking, is to run multiple search processes concurrently. *Concurrent search* performs multiple concurrent backtrack search processes asynchronously on disjoint parts of the *DisCSP* search-space. Each search space includes all variables and therefore involves all agents [35, 9, 37]. One approach to concurrent search was proposed by Hamadi and Bessiere in the *interleaved parallel search algorithm - IDIBT* [9]. IDIBT runs multiple processes of asynchronous backtracking and its multiplicity is fixed at the start of its run [8, 9]. The performance of IDIBT was found to deteriorate for more than 2 contexts (i.e. more than two concurrent ABT processes) [9].

A recent concurrent search algorithm runs multiple backtrack search processes asynchronously. Search processes are initiated and stopped dynamically and this dynamicity was found to enhance the performance of $ConcBT$ and $ConcDB$ to outperform all other $DisCSP$ search algorithms [37, 38]. In *concurrent backtracking* ($ConcBT$), agents pass their assignments to other agents on a *CPA* (Current Partial Assignment) data structure [37]. Each *CPA* represents one search process, and holds the agents' current assignments in the corresponding search process. The search ends unsuccessfully, when all *CPA*s return for backtrack to the initializing agent and the domain of their first variable is empty. The search ends successfully when *one CPA contains a complete assignment*, a value for every variable in the DisCSP.

An agent that receives a *CPA* tries to assign its local variables with values that are not conflicting with the assignments already on the *CPA*, using only the current domains in the search process (*SP*) that is related to the received *CPA*. Any agent can generate a set of CPAs that split the search space of a single CPA that passed through that agent, by splitting the domain of one of its variables. 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 the example in figures 3, 4 and [37, 38]). Splits are performed in search spaces that are traversed heavily and a nice heuristic trigger for splits is the number of steps performed without returning to the starting agent and without finding a solution.

All agents participate in all search processes, assigning their variables and checking for consistency with constraining agents. All search processes are performed asynchronously by all agents, thereby achieving concurrency of computation and shortening the overall time of run for finding a global solution [35]. Agents and variables are ordered randomly on each of the search processes, diversifying the sampling of the search space. Agents generate and terminate search processes dynamically during the run of $ConcBT$, thus creating a distributed asynchronous algorithm [37]. The degree of concurrency during search changes dynamically and enables automatic load balancing.

The best version of concurrent search is Concurrent Dynamic Backtracking ($ConcDB$), that performs dynamic backtracking [7] on each of its concurrent sub-search spaces [38]. Since search processes are dynamically generated by $ConcDB$, the performance of backjumping in one search space can indicate that other search spaces are unsolvable. This feature, combined with the random ordering of agents in each search process, enables early termination of search processes discovered by DB to be unsolvable.

To visualize the main feature of concurrent search, dynamic splitting of search spaces, consider the constraint network that is described in figure 3. All three agents own one variable each, and the initial domains of all variables contain four values $\{1..4\}$. The constraints connecting the three agents are: $X_1 < X_2$, $X_1 > X_3$, and $X_2 < X_3$. The initial state of the network is described on the LHS of Figure 3. In order to keep the example small, no initial split is performed, only dynamic splitting. The number of steps that trigger a split in this example is 4. The first 5 steps of the algorithm run produce the state that is depicted on the RHS of Figure 3. The run of the algorithm during these 5 steps is described in detail below:

1. X_1 assigns 1, and sends a CPA with CPA_steps = 1 to X_2 .
2. X_2 assigns 2, and sends the CPA with CPA_steps = 2, to X_3.
3. X_3 cannot find any assignment consistent with the assignments on the CPA. It passes the CPA back to X_2 to reassign its variable, with CPA_steps = 3.
4. X_2 assigns 3 and sends the CPA again to X_3, raising the step counter to 4.
5. X_3 receives the CPA with X_2's new assignment.

In the current step of the algorithm, agent X_3 receives a CPA which has reached the $steps\_limit$. Before trying to find an assignment for its variable, X_3 sends a split message to X_1 which is the generator of the CPA, and changes the value of the CPA_steps

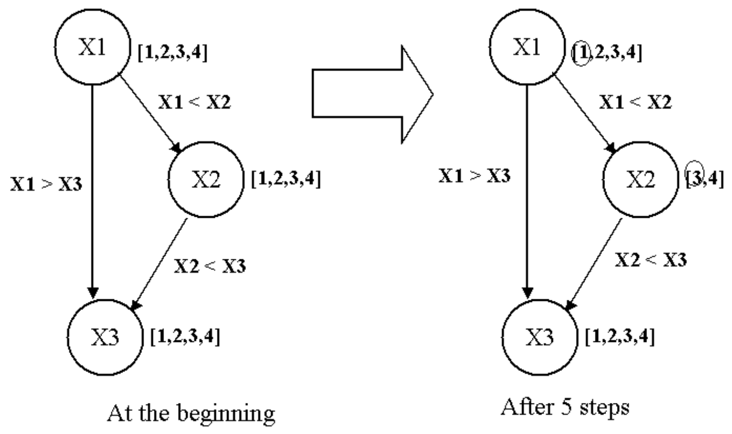


Fig. 3. Initial state and the state after the CPA travels 5 steps without returning to its generating agent

counter to 0. Next, it sends the CPA to X_2 in a backtrack message. When X_1 receives the split message it creates a new search procedure. The new SP has the values 3, 4 in its domain. These values are deleted from the domain of the existing SP. A new CPA is created by X_1, assigned with the value 3 (a value from the new domain) and sent to a randomly chosen agent. Other agents that receive the new CPA create new SPs with a copy of their initial domain.

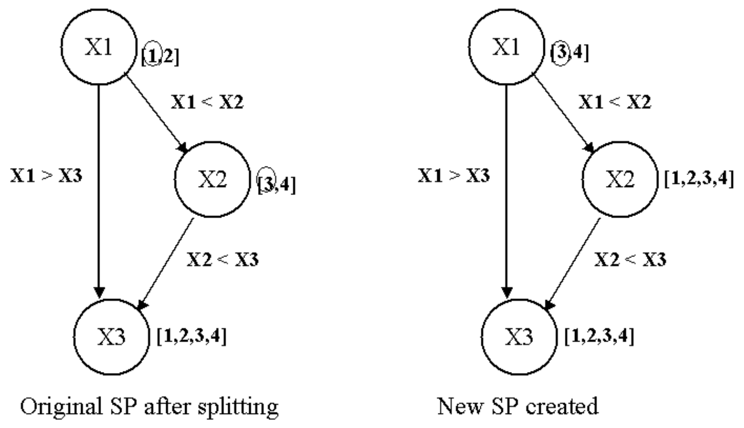


Fig. 4. The new non intersecting search spaces now searched using two different *CPAs*

After the split, two $CPAs$ are passed among the agents. The two CPAs perform search on two non intersecting search-spaces. In the original SP *after the split,* X_1 can assign only values 1 or 2 (see LHS of Figure 4). The search on the original SP is continued from the same state it was in before the split. Agents X_2 and X_3 continue the search using their current domains to assign the original CPA. Therefore, the domain of X_2 does not contain values 1 and 2 which were eliminated in earlier steps and it assigns the value 3 on CPA_1. In the newly generated search space, X_1 has the values $3, 4$ in its domain. Agent X_1 assigns 3 to its variable and the other agents that receive CPA_2 check the new assignment against their full domains (RHS of figure 4).

To see the difference of concurrent search from asynchronous backtracking, let us take the 4-queens example and run $ConcBT$ on it (figures 5 and 6). Three concurrent search prosses (SPs) are started by agent A_1. The three SPs are represented by a triangle, a square and a circle. In the first cycle of computation, agent A_1 splits its domain to three parts and assigns values to the three SPs. These are values 1, 2, and 3. The three $CPAs$ are sent forward to different agents. SP_1 (triangle) is sent to agent A_2, SP_2 (square) is sent to agent A_3 and SP_3 (circle) is sent to agent A_4. Each agent keeps a separate data structure for each SP and computes its assignments, upon receiving a CPA, separately. In the second cycle of computation, agents A_2, A_3, A_4 compute concurrently, each assigning a value to its variable on the CPA it is holding. Each assignment is consistent with all former assignments on the CPA. Agent A_2, for example, assigns the value 3 to CPA_1. Having performed their assignments, all agents send the $CPAs$ to unassigned agents. A_2 sends CPA_1 to agent A_3, agent A_3 sends CPA_2 to agent A_4, and agent A_4 sends CPA_3 to agent A_2.

In cycle 3 again agents A_2, A_3, A_4 are active. Agent A_4 performs a compatible assignment on CPA_2 and sends it further. Agents A_2 and A_3 cannot find a compatible assignment to their variable on CPA_3 and CPA_1 respectively. As a result, they send these two $CPAs$ in a backtrack step. It is easy to follow the next steps of computation in figure 6. In cycle 5 agent A_2 receives two messages. One is from agent A_4 that sent it CPA_3, having revised its assignment from value 1 to value 2. The other message contains CPA_2 (square) with agent A_4's assignment. For clarity of presentation agent A_2 performs the assignments on these two $CPAs$ in two separate cycles of computation. In cycle 5 it assigns CPA_3 with the value 1. In cycle 6 it assigns CPA_2 with the value 4, thus completing a solution. It is important to note that at the same cycle agent A_3 completes another solution concurrently. That of CPA_3.

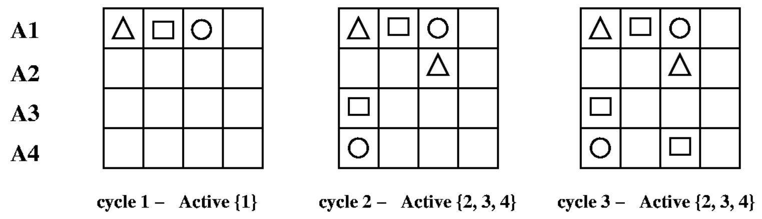


Fig. 5. First 3 cycles of ConcBT - lists of Active Agents in each cycle

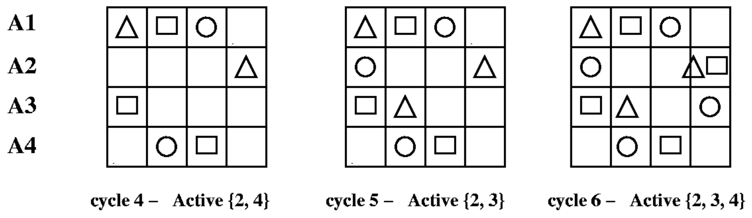


Fig. 6. Cycles 4-6 of ConcBT

2.1 Stochastic search algorithms on DisCSPs

Stochastic search on $DisCSPs$ incorporates agents that perform improvement moves to their local neighbourhoods [32]. The distributed nature of the algorithms relates to the fact that each agent computes steps of improvement to its local neighbourhood and takes into account only its neighbours. The stochastic nature springs from the initial assignment that is random. Like centralized versions of stochastic search, distributed stochastic search runs by improving assignment states until no further improvement is possible or some stopping criterion has been reached [1]. Two important distributed local search algorithms have been studied in the $DisCSP$ literature. Distributed local search algorithm (DSA) [32] and the distributed breakout algorithm (DBA) [28]. The DBA is a natural extension of the breakout algorithm [18] to distributed search on $DisCSPs$.

Distributed stochastic search (DSA) starts by each agent selecting a value randomly for its variable. In each step of the search, each agent sends its assignment to neighbouring agents *if it was changed* at that step, and receives messages from neighbouring agents that changed their assignment [32]. Next, each agent decides whether to keep its value or select another one. New values are selected to achieve the objective of reducing the number of violated constraints. Agents will not change their values if no improvement in the violation of constraints can be achieved. Improvements can be achieved by the use of some strategy for selecting a next value.

There are several strategies for selecting the next value. Selecting values by the use of different strategies (with some probability p) enables DSA to escape from local minima [32]. Strategies range from selecting only values that strictly improve the current state, to selecting values that do not degrade the current state, while there are still constraint violations. The exact step of improvement, among several agents that can potentially improve the value of their neighbourhood, is selected by a decision protocol of all the involved agents. Neighbouring agents propose and compare their potential improvements, selecting the best candidate.

In the distributed breakout algorithm (DBA), a different approach is utilized for escaping local minima. Each constraint is associated with a value and the summation of all constraint violations is used as the cost of a solution. In the process of improving a solution, neighbouring agents exchange messages of possible improvements. Only the agent that can maximaly improve the value is allowed to change its value. Initially all values for all constraints are set to 1. When no improving step can be found, the weights of the violated constraints are increased by 1. This directs the DBA to find improving moves that involve constraints that are persistently violated [33, 18].

In both DSA and DBA agents exchange messages and then select a specific improving move. Thus the algorithms are naturally described in terms of steps or moves. and one can make two observations:

– Each step must be synchronized among all agents, so that the computation of their change of value assignments is based on correct data about their neighbours.
– The "weaker" the condition for permitting a change in DSA, the higher the degree of concurrency of the search process. This is due to larger number of agents performing changes concurrently.

Both of the above observations are related to the efficiency of the algorithm. The policy employed by DSA, for selecting the next assignment, was compared to the Distributed Breakout Algorithm (DBA) [33]. Zhang et. al. have found that on the family of Sensor-CSP problems [2], DSA is superior to DBA. They have also found thrashing behavior of DSA, when the degree of concurrency was too high (i.e. when too "weak" a condition for improvement was used) [33].

The fact that steps of a distributed algorithm are synchronized among agents is also important for assesing its efficiency. It is true that distributed mechanisms for synchronization of actions among agents do exist [11]. Agents can synchronize their steps in a distributed stochastic search, by enumerating the steps and by exchanging special messages that ensure the conclusion of each step (cf. [32]). However, these synchronizing messages must be taken into account when measuring the overall communication load of a distributed search algorithm. Many current studies use a simple simulator for running $DisCSP$ search algorithms [27], thus using synchronized steps of computation at no extra cost. I believe that this experimental setup is quite misleading, as it does not simulate concurrency.

In order to measure correctly the efficiency of a $DisCSP$ algorithm one needs a mechanism for counting concurrent steps or concurrent constraints checks (see section 5). When messages are delayed, any synchronization mechanism causes linear delays in the run of the algorithm (each agent waiting for all others to complete the current step). This will become clearer in section 3, where message delays and their impact on the efficiency of $DisCSP$ search algorithms are discussed in detail. The important point to make is that a correct measure of the efficiency of the existing distributed stochastic search algorithms must incorporate the mechanism for synchronizing search steps explicitly.

Incomplete search algorithms are usually used to find the best solutions [1]. This is natural, since moves of improvement to the overall cost of a solution lead to solutions of lower cost. The above distributed stochastic search algorithms are therefore suitable for finding an optimal solution to a $DisCSP$, on the condition that some overall cost function is defined. A special case of a global cost function can be defined in terms of *valued constraints*. Modi et. al. define a distributed constraints optimization problem (DCOP) as a $DisCSP$ in which all constraints are valued [16]. Valued constraints return a value for each pair of assignments to the constrained variables. The sum of all constraints' values is defined as the overall cost of the solution and the goal of the DCOP is to minimize this cost [16, 17].

A complete and asynchronous search algorithm for DCOP was proposed recently by Modi et. al. [17]. The ADOPT algorithm has agents sending their assignments down the DFS tree in VALUE messages and sending back COST messages. COST messages contain a lower bound and an upper bound to the cost of the subtree rooted at the sending agent. Based on the received COST messages from its children, each agent updates its estimation of the lower and upper bound. Estimations relate to a complete neighbourhood of each agent. COSTs from its children combined with the computed cost of its constraints with its (constraining) ancestors. The central point of the ADOPT algorithm is its ability to maintain bounds asynchronously [17]. It does so by using thresholds on cost that agents send down the DFS tree. Thresholds are based on the cost estimation in each agent and are split heuristically among its children. Thresholds improve the efficiency of the search by pruning assignments that result in costs higher than the threshold. The algorithm is distributed, asynchronous and complete and is shown to terminate deterministically upon arrival at an optimal solution [17].

3 Message Delays

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. In order to investigate the impact of message delays on DisCSP algorithms, two essential requirements have to be satisfied:

- Means of controling the amount and type of delays in the experimental setup.
- A common scale for message delays and the performance measures of distributed search algorithms.

The first study of the impact of message delays on $DisCSP$ algorithms used randomly generated delays that were measured in real time of runs [6]. The results indicated a strong deterioration in the performance of ABT with random message delays. However, the scale of delays dictated the measurement of performance in real time. While this is acceptable, it is highly implementation dependent. As explained in section 1, the performance of distributed algorithms is measured by two standard means that are implementation independent. To achieve such measurement for $DisCSP$ algorithms, one must use a well controlled environment in the form of a simulator. 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 [10, 13, 22]. 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 [10].

On systems with message delays, the situation is more complex. For the simplest possible algorithm, Synchronous Backtrack (SBT) [27], 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 [39].

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 [13]. 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. 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. This generates the value of the global clock (of the Mailer) which is the largest of all logical times of all agents. Next, a delay for the message (in number of steps) is selected. 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 [39]. 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 $outgoing\_queue$. Finally, the $Mailer$ 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) [27], every delay needs the mechanism of updating the Mailer's LTC.

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 [10], as proposed for DisCSPs by [13], and extended here for message delays. A sample of results of runs with random message delays, for algorithms described in section 2, is given in section 5.

4 Privacy of DisCSP search algorithms

An important goal of search algorithms for the distributed constraint satisfaction problem is to support agents' privacy. During cooperative search for a globaly consistent solution, agents exchange messages about their assignments and about conflicts with other agents' assignments. This creates a natural trade-off between information disclosure and the efficiency (and correctness) of the distributed search process. The first to investigate measures of privacy for DisCSPs were Meseguer and Jimenez [15]. They presented separate algorithms for maintaining two types of privacy during the run of the asynchronous backtracking (ABT) algorithm. One for maintaining privacy of values, where agents do not disclose their assignments, and one for maintaining privacy of constraints.

Privacy of constraints occurs when agents keep part of the constraints between them and a constraining agent private. A vivid example is to have a problem in which two agents, one acting like a queen on a chessboard and the other like a knight, have to find a consistent assignment for themselves. Each one knows what positions of the other agent are forbidden by its own type of moves. However, if the agents keep their constraints private, none of them knows which positions are forbidden by the types of moves of the other agent [5]. It turns out that both types of privacy can be maintained simultanously by an interesting version of asynchronous backtracking. It involves two runs of the ABT algorithm, in two opposite directions. This privacy keeping version of ABT was proposed by Brito and Meseguer in 2003 [5].

Securing complete privacy for two agents exchanging information by messages can be also tackled by the use of secure protocols. This approach has been proposed by Yokoo et. al. in [30]. In their work, Yokoo et. al. use an additional set of agents that act as trusted parties for exchanging assignment proposals and constraints checks among the $DisCSP$ agents. Thus, a standard method of third party secure exchange of information is established [30]. No investigation was reported, to try and find the added cost, in both computation and network load, due to such a secure asynchronous search protocol.

A different approach for investigating the privacy of distributed search was presented first by Wallace and Freuder [26]. The idea is to investigate the trade-off between privacy loss and the efficiency of distributed search by using a specific scenario of a distributed CSP. The selected family of distributed search problems was that of Scheduling Meetings among multiple agents. This concrete family of problems was used to compare the amount of needed computations for finding a solution, when different quantities of information were exchanged among the searching agents. The meeting scheduling problem (MSP) was studied in a very restricted form in [26]. The tradeoff between the privacy of agents' meetings (as appear in their own calendar) and the efficiency of the search process was studied by using instances of MSPs that have only one meeting to coordinate, which all agents have to attend [26]. Agents must be able to get from their private meetings to the scheduled meeting according to the traveling time constraints. Each agent has its own private calendar that defines its constraints regarding the time and location of the meetings.

In [25], the family of MSPs as been extended to be the graph coloring problem for n meetings (variables). The problem is to assign time-slots to all n variables (meetings), such that each variable is owned by more than one agent. The constraints among the values assigned to meetings which include a specific agent are inequality constraints. This creates a graph coloring problem of a distributed nature. Each agent owns the variables corresponding to meetings in which it participates and an inequality constraint holds among them [25]. However, the investigation of privacy-efficiency trade-off in [25] uses "synchronous distributed backtracking" for solving the problem. It can be shown that for asynchronous search algorithms, such as ABT, the notion of privacy is much more difficult to establish [12].

To overcome the above problem an investigation of the general MSP problem (with general arrival-time constraints) that uses an enhanced version of the asynchronous backtracking (ABT) algorithm was recently performed [12]. It measures the effect of asynchronous exchange of additional information on asynchronous search, to find that volunteering information by agents reduces search effort. The main idea is to volunteer additional information during the sending back of $Nogoods$. Additional $Nogoods$ reduce the amount of messages sent and the number of concurrent steps of computation [12]. This mechanism has the nice quality that $Nogoods$ remain valid and can retain their relevance all through the asynchronous search process.

For the general MSP the issue of privacy is problematic because the two existing approaches in the $DisCSP$ literature cannot be extended in a natural way. The study of privacy trade-off [26, 25] uses the refusals to a proposed meeting, to construct a "shadow schedule". Thus, accumulating information on the schedules of other agents. This cannot be done for the general MSP, where conflicts arise from meetings that are being scheduled asynchronously. The other approach to privacy enables to keep the privacy of constraints and assignments [5]. For the general MSP, any proposed meeting is by definition equal to the proposer's assignment. All inter-agent constraints are equality constraints (i.e. meeting at the same time and the same place). This rules out keeping assignments private, they are constrained to be equal. Unfortunately, this also rules out the standard privacy of constraints, in which constraints between two agents are split among the agents [5]. Equality constraints are symmetric and keeping them private to one of the agents that participate in a meeting does not seem to make any sense [12].

5 Experimental Evaluation

The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance - time, in the form of steps of computation [11, 27], and communication load, in the form of the total number of messages sent [11]. Comparing the number of concurrent steps of computation of search algorithms on DisCSPs, measures the time of run of the algorithms.

Concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [10, 13]. 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 close to Lamports logical time [10]. If instead of steps of computation we count the number of concurrent constraints check peformed (CCCs), we take into account the local computational effort of agents in each step [13].

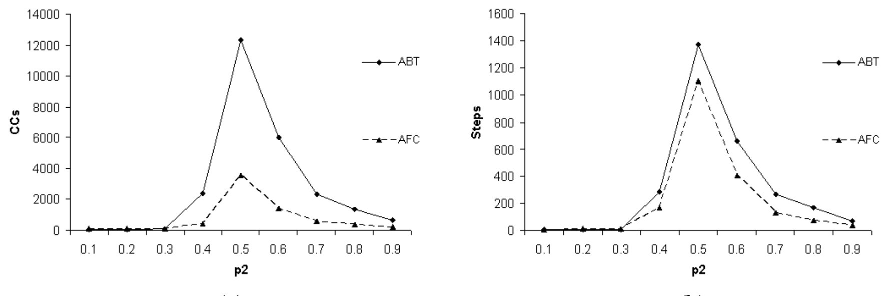
An important part of the experimental evaluation is to measure the impact of imperfect communication on the performance of concurrent search. Message delay can change the behavior of distributed search algorithms [6]. In the presence of concurrent computation, the time of message delays must be added to the total algorithm time *only if no computation was performed concurrently*. To achieve this goal, we use a simulator which counts message delays in terms of computation steps and adds them to the accumulated run-time when no computation is performed concurrently [39].

Experiments were conducted on random networks of constraints of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of CSP algorithms cf. [21, 23]). All sets of experiments were conducted on networks with either 15 or 10 agents ($n = 10, 15$) and 10 values for each variable ($k = 10$). Two values of constraints density were used in different experiments, $p_1 = 0.4$ and $p_1 = 0.7$. The tightness value p_2, is varied between 0.1 and 0.9, to cover all ranges of problem difficulty.

5.1 Asynchronous forward-checking vs. ABT

The performance of AFC is compared to Asynchronous BackTracking (ABT) [27]. In our implementation of ABT, the $Nogoods$ are resolved and stored according to the method presented in [3]. Based on Yokoo's suggestions [28] the agents read, in every step, all messages in their mailbox before performing computation.

Figure 7 presents a comparison of the computational effort performed by AFC and ABT on randomly generated $DisCSPs$. The advantage in concurrent constraints checks (figure 7(a)) of AFC over ABT is more pronounced then in concurrent steps (figure 7(b)). This indicates that the distributed procedure which maintains local consistency in AFC is efficient. It needs fewer constraints checks per computation step on the average. The communication load, as measured by the total number of messages sent during search, is also lower for AFC than for ABT (see [14]).

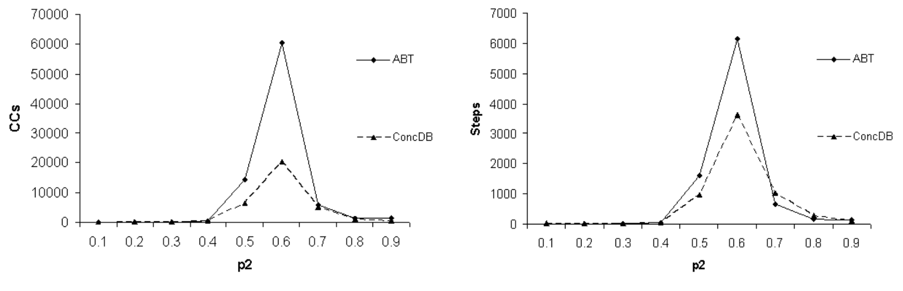


(a) (b)

Fig. 7. (a) Number of concurent constraints checks in AFC, and ABT, (b) Number of concurrent steps for both algorithms.

5.2 ConcDB vs. Asynchronous Backtracking

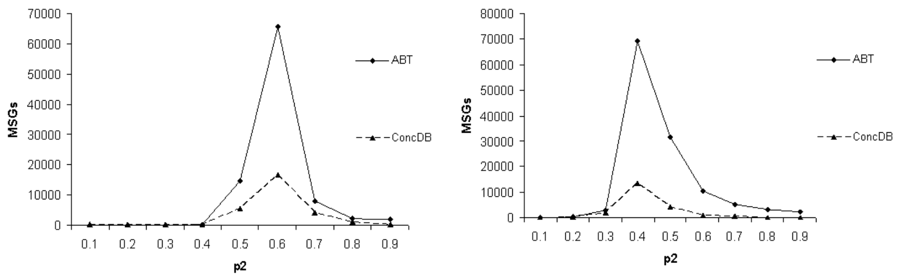
The performance of concurrent dynamic backtracking (*ConcDB*) can be compared to asynchronous backtracking (ABT) [27]. In ABT agents assign their variables asynchronously, and send their assignments in $ok?$ messages to other agents to check against constraints. A fixed priority order among agents is used to break conflicts. Agents inform higher priority agents of their inconsistent assignment by sending them the inconsistent partial assignment in a $Nogood$ message. Our implementation of ABT is the same is in the above comparison to AFC [3]. Based on Yokoo's suggestions [27] the agents read, in every step, all messages received before performing computation.



(a) (b)

Fig. 8. (a) Number of concurrent constraints checks performed by ConcDB and ABT, (b) Number of concurrent steps for both algorithms.

The LHS of figure 8 presents the comparison of the number of concurrent constraints checks performed by $ConcDB$ and ABT on problems with low dinsity ($p_1 = 0.4$). For the harder problem instances, *ConcDB* outperforms ABT by a factor of 3. On the RHS of figure 8 The results are presented in the number of concurrent steps of computation. The smaller factor of difference can be related to the larger amount of local computation ABT perfoms in each step since it reads all the messages which it received up to this step.



(a) (b)

Fig. 9. Total number of messages sent by ConcDB and ABT on DisCSPs with low density (a) and high density (b).

When it comes to communication load, the advantage of concurrent search over asynchronous backtracking is very pronounced. As can be seen in figure 9, for harder problem instances ABT *sends 4 times more messages than ConcDB*. For higher problem density (figure 9(b)) the factor is even higher.

5.3 The impact of message delays

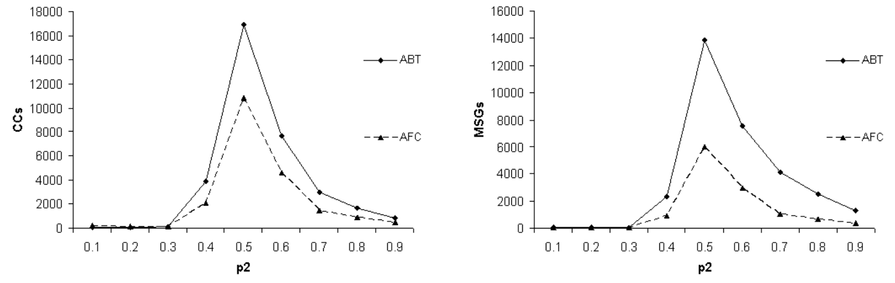
In another set of experiments, each message was delayed by a random number of logical concurrent constraints checks, between 5 and 15. The AFC algorithm was compared to ABT. The results in figure 10 show as expected, that the performance of both algorithms deteriorates with message delays. The difference in concurrent constraints checks, between AFC and ABT is smaller on systems with random message delay. The difference in the total number of messages however, is larger (RHS of figure 10). In both cases AFC has an advantage over ABT.

When messages are delayed ABT cannot read multiple messages before performing computation. Therefore the actual steps it performs and the number of messages it sends grow by a significant factor. AFC is slowed down by message delay when the CPA is delayed, or a Not_OK message indicating a need for backtrack is delayed, while the CPA is moving forward. In other words, message delay disables some of the pruning of AFC. It delays the backtrack triggered by forward checking.

In contrast to both AFC and ABT, concurrent search is quite robust to message delays. Figure 11 presents the results of the set of experiments in which ConcDB and ABT were run on a system with random message delay. Each message was delayed between 5 to 10 steps and the results in logical steps are presented for low and high density ($p_1 = 0.4$ on the LHS and $p1 = 0.7$ on the RHS of figure 11). Random message delay deteriorates the performance of asynchronous backtracking while the effect on concurrent dynamic backtracking is minor. The results in figure 11 show a larger factor of difference between the two algorithms.

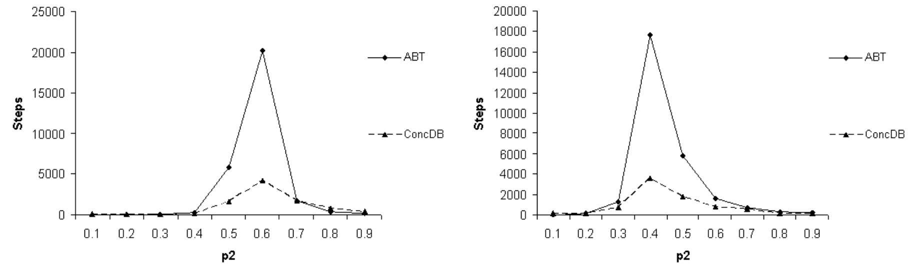
6 Discussion

Distributed constraints satisfaction problems (DisCSPs) are composed of agents that own parts of the CSP. In a DisCSP search algorithm, all agents cooperate in search for a



(a) (b)

Fig. 10. (a) Number of logical concurrent steps performed by AFC and ABT on DisCSPs with random message delays, (b) Total Number of messages sent with random message delays.



(a) (b)

Fig. 11. (a) Number of logical concurrent steps performed by ConcDB and ABT on low density DisCSPs, (b) on high density DisCSPs.

globally consistent solution to the problem. Since agents own variables, DisCSP algorithms involve assignments of values to variables by agents and communication among agents in order to arrive at a consistent solution. DisCSPs can serve as a general model for distributed problem solving - from timetabling a set of departments to cooperative search on a graph.

Two general families of search algorithms on DisCSPs have been presented. One family maintains a single search process at all times and the other family has multiple concurrent search processes. The classical single search algorithm is asynchronous backtracking (ABT) and a recent member of this family is asynchronous forward-checking (AFC). Both have been presented in section 2 in their updated versions. ABT reads all messages at each step and resolves Nogoods [3] and AFC backtracks by maintaining (shortly) obsolete search processes [14, 19]. A very successful multi search process algorithm is concurrent dynamic backtracking ($ConcDB$). ConcDB splits the search space dynamically during search and utilizes interaction among search spaces [38]. The interaction results from backjumping, that abolishes adjacent sub search spaces (see section 2).

A major issue of DisCSP algorithms is that of distributed efficiency measures. Asynchronous algorithms cannot be measured simply by the total number of computational operations that are performed by all agents. The best proposed measure to date is to compute the number of concurrent computation steps or the number of concurrent constraints checks (CCCs) [13]. This measure generalizes Lamport's idea of clock synchronization [10] to the case of Constraints checks for asynchronous DisCSP search algorithms. Our experimental evaluation used these concurrent performance measures for DisCSP algorithms on randomly generated problems.

An extensive experimental evaluation of all complete $DisCSP$ algorithms has been presented. The experimental behavior of the multi search algorithm on random DisCSPs clearly indicates its efficiency, compared to algorithms of a single search process like ABT. Experiments were conducted for different constraints densities, a wide range of constraints tightness and in systems with random message delays. In all experiments and for three different measures of performance, $ConcDB$ outperforms ABT by a large margin.

The delay of messages can have a strong impact on the efficiency of distributed search algorithms on $DisCSPs$ [6]. The study of the behavior of $DisCSP$ algorithm under message delays was presented as part of the experimental evaluation (section 5). 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. 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).

The results presented in section 5 strengthen the results reported by [6], and do so for a larger family of random problems. In contrast, the impact of random message delays on concurrent search algorithms is minor. This is very apparent in Figure 11, where the number of steps of computation of $ConcDB$ is lower than that of ABT by a factor of 4.5 in the presence of message delays (compared to a factor of 3 only, with no message delays in figure 8(b)). 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 concurrent algorithms is much smaller than the load of asynchronous backtracking and is not affected by message delays.

The trade-off between privacy and efficiency for $DisCSP$ algorithms was investigated by Wallace [26]. Two partial privacy-keeping versions of ABT were proposed by [5]. These investigations leave much room for future research, since privacy is a central issue to distributed search. In section 4 a recent result that uses the meeting scheduling problem was described [12]. For MSPs, volunteering additional information (in the form of $Nogoods$) improves the performance of ABT. This generalizes former findings of [26], for general MSPs. An interesting generalization of this result is that volunteering additional information can enhance the efficiency of asynchronous search *on general DisCSPs*.

Acknowledgement: My deepest thanks to Roie Zivan, whose work with me on DisCSP search algorithms and on the impact of message delays has triggered this Turotial.

References

1. Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Son, Chichester, 1997.
2. R. Bejar, B. Krishnamachari, C. Gomes, and B. Selman. Distributed constraint satisfaction in a wireless sensor tracking system. In *Workshop on Distributed Constraint of IJCAI01*, 2001.
3. C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence (to appear)*, 2004.
4. C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Workshop on Distributed Constraint of IJCAI01*, 2001.
5. I. Brito and P. Meseguer. Distributed forward checking. In *Proc. CP-2003*, pages 801–806, September/October Irland, 2003.
6. C. Fernandez, R. Bejar, B. Krishnamachari, and K. Gomes. Communication and computation in distributed csp algorithms. In *Proc. CP2002*, pages 664–679, Ithaca NY USA, July 2002.
7. M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.
8. Y. Hamadi. Distributed interleaved parallel and cooperative search in constraint satisfaction networks. In *Proc. IAT-01*, Singappore, 2001.
9. Y. Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, 2002.
10. L. Lamport. Time, clocks, and the ordering of events in distributed system. *Communication of the ACM.*, 2:95–114, April 1978.
11. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.
12. A. Meisels and O. Lavee. Using additional information in discsp search. In *Proc. Workshop on Distributed Constraint Reasoning, DCR-04*, Toronto, September 2004.
13. 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*, Bologna, July 2002.
14. A. Meisels and R. Zivan. Asynchronous forward-checking for distributed csps. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2004.
15. P. Meseguer and M. A. Jimenez. Distributed forward checking. In *Proc. CP-2000 Workshop on Distributed Constraint Satisfaction*, Singapore, September 2000.
16. P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2003.
17. P. J. Modi, W. M. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 2004.
18. Paul Morris. The breakout method for escaping from local minima. In *Proc. AAAI-93*, pages 40–45. AAAI Press/MIT Press, 1993.
19. V. Nguyen, D. Sam-Haroud, and B. Faltings. Distributed dynamic backtracking. In *Proc. CSCLP 2004: Workshop of ERCIM/CoLogNet*, Lausanne, June 2004.
20. A. Petcu and B. Faltings. A value ordering heuristic for local search in distributed resource allocation. In *CSCLP04*, Lausanne Switzerland, Febuary 2004.
21. P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
22. M. C. Silaghi. *Asynchronously Solving Problems with Privacy Requirements*. PhD thesis, Swiss Federal Institute of Technology (EPFL), 2002.
23. B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

24. G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, New Hamphshire, October 1996.
25. R. J. Wallace. Reasoning with possibilities in multiagent graph coloring. In *4th Intern. Workshop on Distributed Constraint Reasoning*, pages 122–130, 2003.
26. R. J. Wallace and E. C. Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. In M. Yakoo, editor, *AAMAS-02 Workshop on Distributed Constraint Reasoning*, pages 176–182, Bologna, 2002.
27. M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.
28. M. Yokoo. *Distributed Constraint Satisfaction Problems*. Springer Verlag, 2000.
29. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.
30. M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP-2002, 8th International Conference, CP 2002, Ithaca, NY, USA*, pages 387–401, september 2002.
31. W. Zhang and L. Wittenburg. Distributed breakout revisited. In *AAAI-2002*, Edmonton Alberta Canada, 2002.
32. W. Zhang and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Workshop on Probabilistic Approaches in Search AAAI-2002*, pages 53 – 59, Edmonton Alberta Canada, July 2002.
33. W. Zhang and Z. Xing. Distributed breakout vs. distributed stochastic: A comparative evaluation on scan scheduling. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 192 – 201, Bologna Italy, July 2002.
34. W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *Proc. AAMAS-2003*, pages 185 – 192, Melbourne Australia, July 2003.
35. R. Zivan and A. Meisels. Parallel backtrack search on discsps. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, Bologna, July 2002.
36. R. Zivan and A. Meisels. Synchronous vs asynchronous search on discsps. In *Proc. European Workshop on Milti Agent System EUMAS*, Oxford, December 2003.
37. R. Zivan and A. Meisels. Concurremt backtrack search for discsps. In *Proc. FLAIRS-04*, Maiami Florida, May 2004.
38. R. Zivan and A. Meisels. Concurrent dynamic backtracking for distributed csps. In *CP-2004*, Toronto, 2004.
39. R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *submit to DCR-04*, Toronto, 2004.