Synchronous vs Asynchronous search on DisCSPs

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 variables, that are connected by constraints to variables of other agents. There are two known measures of performance for distributed search - the computational effort which represents the total search time and the number of messages sent which represents the network load. Due to the distributed nature of the problem, the behavior of the experimental environment is extremely important. However, most experimental studies have used a perfect simulator with instantaneous message delivery. The present paper investigates two families of distributed search algorithms on DisCSPs, Synchronous and Asynchronous search. Improved versions of the two families of algorithms are presented and investigated. The performance of the algorithms of these two extended families is measured on randomly generated instances of DisCSPs. The results of the investigation are twofold. First, the delay of messages is found to deteriorate the performance of asynchronous search by a large margin. This shows that a correct (and realistic) experimental scenario is important. Second, when messages are delayed, synchronous search performs better than asynchronous search in terms of computational effort as well as in network load. It turns out that asynchronous search fails to use its multiple computing power to an advantage.

1 Introduction

Distributed constraints satisfaction problems (*DisCSP*s) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [Yokoo2000,Solotorevsky et. al.1996]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them, to check consistency of their proposed assignments against constraints with variables that belong to different agents [Yokoo2000,Bessiere et. al.2001].

Several asynchronous search algorithms on DisCSPs have been proposed in recent years [Yokoo et. al.1998,Bessiere et. al.2001,Silaghi et. al.2001]. All of these algorithms process assignments of agents asynchronously and rely on Nogoods for their correctness and termination. In asynchronous search agents perform assignments asynchronously and send out messages to constraining agents, informing them about their assignments. Due to the asynchronous nature of agents' operations, the global assignment state at any particular instance during the run of an asynchronous search algorithm

is in general inconsistent. The motivation in using asynchronous search was that by allowing agents to perform concurrently and independent of one another, the distributed nature of the system will be used to gains in efficiency [Yokoo et. al.1998].
The immediate drawback is the network load. Agents performing computations concurrently and performing assignments, load the network with messages informing other agents of their actions. Not many comparisons were made between the efficiency of synchronous and asynchronous search algorithms on DisCSPs. In [Yokoo2000] the Asynchronous Backtrack algorithm (ABT) was compared with Synchronous Backtrack (SBT) on the n-queens problem and was found to arrive at a solution within a smaller number of cycles. There are several important changes that have the potential of improving the comparative study of [Yokoo2000]. These improvements relate to the following features of the former study:

1. The synchronous algorithm used (SBT) is a distributed version of chronological backtrack (CBT), which is the slowest synchronous search algorithms (compared to Conflict based Backjumping (CBJ) for example) [Prosser1993].
2. n-queens problems have a large number of solutions. It is interesting to perform the comparison for harder problem instances [Smith1994].
3. The experiments were performed on a simulator with instantaneous messages. The performance of ABT was found to be highly dependent on the communication quality of the network [Fernandez et. al.2002].

The main goal of the present paper is to look closer at the performance of members of the two families of search algorithms, synchronous and asynchronous, on $DisCSPs$. In order to rectify the second point on our list of experimental drawbacks, the present paper compares the performance of synchronous and asynchronous algorithms on a range of randomly generated instances of DisCSPs that includes solvable, unsolvable, and phase transition problems [Smith1994]. To create a fair comparison of asynchronous and synchronous algorithms, one needs to note some specific features of ABT.

- The detection of an inconsistent partial assignment determines a conflict set which leads to a backjump to the culprit agent [Yokoo et. al.1998,Bessiere et. al.2001].
- Eliminated values are returned to variables domains only after the eliminating $Nogood$ becomes inconsistent with the agent view.

These properties of ABT are not unique to asynchronous search (cf. [Ginsberg1993]), and by using them in synchronous algorithms one can decrease the computational cost (see Section 4).

The vast majority of experiments on DisCSP search were performed on a simulator [Yokoo2000] and proceeded in cycles or rounds. This experimental setup is actually equivalent to having messages that are instantaneous. At each round of the simulator all messages of the former round arrive at their destination [Yokoo2000]. When all messages arrive within one cycle, agents always perform computations that are based on an accurate view of the state of all other agents. When the delay of messages is large enough, there is some inconsistency between the agent's view of the system state and the 'real' current state of the system, which causes a waste of computational effort. Needless to say that this particular experimental setup is out of tune with an asynchronous

environment and algorithm. The experimental evaluation presented in this paper uses instances of $DisCSPs$ with a wide range of difficulties and is performed with different communication delays.

For a performance measure of the concurrent computational effort performed by agents, one needs a better parameter than the number of cycles, since the computational effort performed by different agents, in different cycles, is not necessarily identical. One such measure, which is independent of implementation details, is the number of concurrent constraints checks [Meisels et. al.2002]. The results show that the performance of synchronous search is similar to that of asynchronous search, in terms of computational effort. Moreover, on systems with fixed and random message delay the computational effort of synchronous search does not change while for asynchronous search the computational cost grows by a large factor. Distributed constraint satisfaction problems ($DisCSPs$) are presented in section 2. A description of the different versions of ABT and comparisons between them on systems with different communication qualities, are presented in section 3. A description of synchronous backtracking (SBT) algorithm and of an improved version, $SynCBJ$, is presented in section 4. Expeirimental results, comparing the synchronous algorithms with ABT are also included in section 4. A discussion of the performance and advantages of the algorithms on different $DisCSP$ instances and communication networks, is presented in section 5. Our conclusions are presented in section 6.

2 Distributed Constraint Satisfaction

A distributed constraints network (or a distributed constraints satisfaction problem - *DisCSP*) is composed of a set of k agents $A_1, A_2, ..., A_k$. Each agent A_i contains a set of constrained variables $X_{i_1}, X_{i_2}, ..., X_{i_{n_i}}$. Constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables. For a set of constrained variables $X_{i_k}, X_{j_l}, ..., X_{m_n}$, with domains of values for each variable $D_{i_k}, D_{j_l}, ..., D_{m_n}$, the constraint is defined as $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. A **binary constraint** R_{ij} between any two variables X_j and X_i is a subset of the Cartesian product of their domains; $R_{ij} \subseteq D_j \times D_i$. In a distributed constraint satisfaction problem *DisCSP*, the agents are connected by constraints between variables that belong to different agents (cf. [Yokoo et. al.1998,Solotorevsky et. al.1996]). In addition each agent has a set of constrained variables, i.e. a *local constraint network*. An assignment (or a label) is a pair $< var, val >$, where var is a variable of some agent and val is a value from var's domain that is assigned to it. A *compound label* is a set of assignments of values to a set of variables. A **solution** P to a *DisCSP* is a compound label that includes all variables of all agents, that satisfies all the constraints. Following all former work on *DisCSP*s, agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. An agent can send messages to any one of the other agents. The delay in delivering a message is assumed to be finite [Yokoo2000]. One simple form of messages for checking constraints, that appear in many distributed search algorithms, is to send a proposed assignment $< var, val >$, of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of

its variables and returns a message that either acknowledges or rejects the proposed assignment (cf. [Yokoo et. al.1998,Bessiere et. al.2001]). The following assumptions are routinely made in studies of $DisCSP$s [Yokoo2000,Bessiere et. al.2001]. and are assumed to hold in the present study.

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.

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. [Prosser1994,Smith1994]). Experiments were conducted on networks with 10 variables ($n = 10$) and 10 values ($k = 10$). All instances were created with density parameter $p1$ set to 0.7 ($p1 = 0.7$). The value of $p2$ varied between 0.1 to 0.9. In order to evaluate the algorithms performances, it is best to compare two independent measures of performance search effort in the form of constraints checks and communication load in the form of the total number of messages sent. By measuring computational effort in the number of constraints checks performed instead of the number of cycles, we take in to account the possibility of agents performing different amount of computational effort in different cycles. The evaluation of the computational effort of distributed algorithms has to take concurrency into account. We count the concurrent effort made by agents using the method of counting concurrent constraint checks (CCCs) [Meisels et. al.2002] in which each agent holds a counter of constraints checks it makes. Every message sent 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 close measure to the accurate cost of the concurrent effort during the run of the algorithm.

3 Asynchronous Backtrack algorithms

The $Asynchronous\ Backtrack\ algorithm$ (ABT) was presented in several versions over the last decade [Yokoo1992,Yokoo et. al.1998,Yokoo2000,Bessiere et. al.2001]. In the ABT algorithm, agents hold an assignment for their variables at all times, which is consistent with their view of the state of the system. 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. It then sends back a Nogood which is based on its former inconsistent $Agent\_view$ and makes another attempt to assign its variable [Yokoo2000,Bessiere et. al.2001]. This is in contrast to synchronous search, in which an agent that cannot find a consistent assignment sends a backtrack

message and waits for the state of the system to change before it attempts to reassign its variable. The following descriptions of ABT, and its different versions, all have the above property.

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

- **when received** $(\textbf{nogood}, x_j, nogood)$ **do**
  1. add nogood to nogood list;
  2. **when** $nogood$ contains an agent x_k that is not its neighbor **do**
  3. request x_k to add x_i as a neighbor,
  4. and add (x_k, d_k) to $agent\_view$; **end_do**;
  5. $old\_value \leftarrow current\_value$; **check_agent_view**;
  6. **when** $old\_value = current\_value$ **do**
  7. send (**ok?**, $(x_i, current\_value)$) to x_j ; **end_ do**; **end_do**;
- procedure **check_agent_view**
  1. **when** $agent\_view$ and $current\_value$ are not consistent **do**
  2. **if** no value in D_i is consistent with $agent\_view$ **then backtrack**;
  3. **else** select $d \in D_i$ where $agent\_view$ and d are consistent;
  4. $current\_value \leftarrow d$;
  5. send (**ok?**,(x_i, d)) to $low\_priority\_neighbors$; **end_if**;**end_do**;
- procedure **backtrack**
  1. $nogood \leftarrow agent\_view$;
  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_i 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. 1. ABT algorithm with full $Nogood$ recording

3.1 Asynchronous Backtrack ABT with full nogood recording

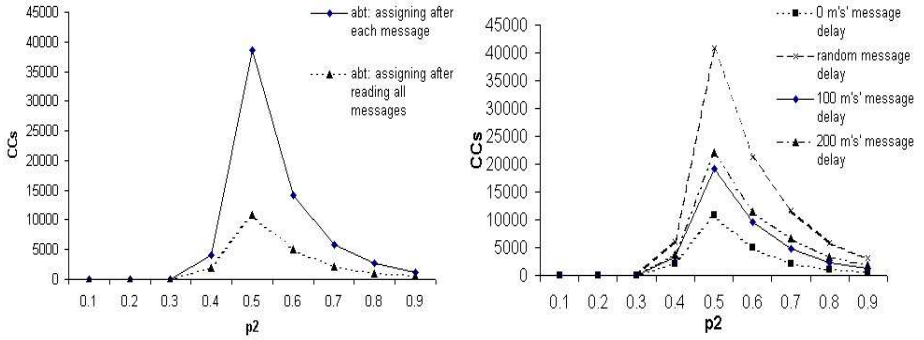
The Asynchronous Backtrack algorithm ABT [Yokoo2000], has a total order of priorities among agents. Agents have a data structure, called $Agent\_view$, which contains the most recent assignments received from agents with higher priority. The algorithm starts by each agent assigning its variable, and sending the assignment to neighbor agents with lower priority. When an agent receives a message containing an assignment (an $ok?$ message [Yokoo2000]), it updates its $Agent\_view$ with the received assignment and if needed replaces its own assignment, to achieve consistency. Agents that reassign their variable, inform their lower priority neighbors by sending them $ok?$ messages. Agents that cannot find a consistent assignment, send the inconsistent tuple in their $Agent\_view$ in a backtrack message (a $Nogood$ message [Yokoo2000]). In the simplest form of the ABT algorithm, the complete $Agent\_view$ is sent as a $Nogood$. The $Nogood$ is sent to the agent with the lowest priority whose assignment is included in the $Nogood$.

Agents that receive a $Nogood$, check its relevance against the content of their $Agen\_view$. 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 resending it an $ok?$ message with its assignment. An agent A_i which receives a $Nogood$ containing an assignment of an agent A_j which is not included in its $Agent\_view$, adds the assignment of A_j to the $Agent\_view$ and sends a message to A_j asking it to add a link between them i.e. inform A_i about all assignment changes it performs in the future.

After an agent A_i sends a $Nogood$ message to agent A_k, it removes the assignment of A_k from its $Agent\_view$ and tries to reassign its variable. This means that if agent A_i still cannot find a consistent assignment for its variable, another $Nogood$ will be created, sent to an agent with higher priority than A_k and its assignment will again be removed from A_is $Agent\_view$ before A_i makes another attempt to assign its variable. This process will continue until A_i succeeds to find a consistent assignment or its $Agent\_view$ empties and a $no\_solution$ is declared. The code of the ABT algorithm is presented in figure 1.

3.2 Improvements to ABT

The first improvement of the performance of ABT requires agents to read all messages they receive before performing computation [Yokoo et. al.1998]. A formal protocol for such an algorithm was not published. The idea is not to perform the procedure $check\_agent\_view$ until all the messages in the agent's 'mailbox' are read and the $Agent\_view$ is updated. Figure 2 (a) presents the difference in computation effort, between a version of ABT performing an assignment after each message received and a version of ABT reading all messages before performing an assignment.



(a) (b)

Fig. 2. (a) Number of concurrent constraints checks performed by ABT performing assignments after each message received and performing assignments after reading all messages, (b) Number of concurrent constraints checks performed by ABT running on networks with different forms of communication.

Although it is clear that a version of ABT that reads all messages received in every step of computation can be much more efficient, such a property makes the efficiency of

ABT dependent on the form of communication. The consistency of the $Agent\_view$ held by an agent, with the actual state of the system before it begins the assignment attempt is affected directly by the number and the relevance of the messages it received up to this step. Figure 2 (b) presents the performance of ABT, solving the same $DisCSPs$, on systems with different message delay including the results of ABT running on a DisCSP with random delay of values between 100 and 300 milliseconds.

In our results, as in [Fernandez et. al.2002], ABT performs worse in a random delay system. These results are not surprising since for fixed delays an agent receives messages from all of its neighbors at about the same time, and can in most cases perform the assignment while holding most of the relevant information. When the messages arrive after a random delay, agents are more likely to respond to a single message, which tends to deteriorate the algorithm's performance. The most important conclusion from Figures 2 is that the improvement that results from reading all incoming messages in each cycle is washed out completely when messages have random delays. This was also a major experimental result of [Fernandez et. al.2002], but, did not have a clear explanation there.

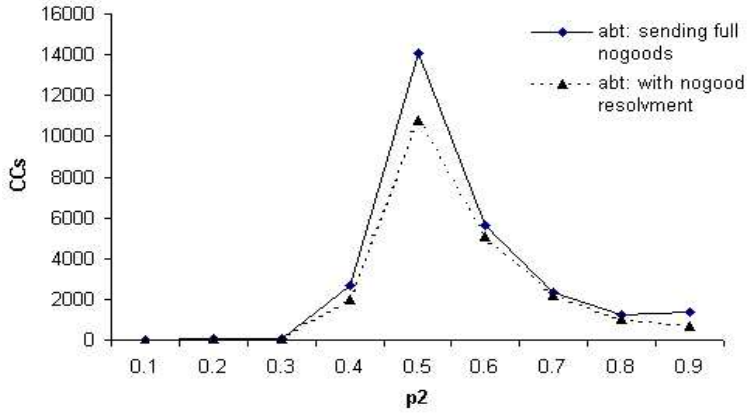


Fig. 3. Number of concurrent constrains checks performed by ABT sending full Nogoods and ABT resolving conflicting Nogoods.

The simplest form of ABT sends $Nogood$s that include the complete $Agent\_view$ of the sending agent. Although the algorithm instructs the agent to send back all inconsistent subsets of the $Agent\_view$, [Yokoo et. al.1998,Yokoo2000] explains that such a search for inconsistent subsets is computationally expensive. Consequently, it is enough to send the complete content of the $Agent\_view$. The reason for this computation not to be essential can be shown by the following description. Assume agent A_i sends back a $Nogood$ to agent A_j, removes the assignment of A_j from its $Agent\_view$ and makes another attempt to reassign its variable. If a shorter inconsistent assignment still exists in its $Agent\_view$, another $Nogood$ will be sent to agent A_{j-1} (assuming agent A_{j-1} is a neighbor of agent A_i). Let A_l be the lower priority agent in the $Agent\_view$ of A_i, such that the removal of A_l's assignment enables A_i to assign its variable. Then at the end of the above process a $Nogood$ message will be sent to A_l. In

– **SBT**:
  1. done ← false
  2. **if**(first_agent)
  3. CPA ← create_CPA
  4. assign_CPA
  5. **while**(**not** done)
  6. **switch** msg.type
  7. $stop$: done ← true
  8. $backtrack$: remove_last_assignment
  9. CPA **or** $backtrack$: 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. **if**(first_agent)
  2. CPA ← no_solution
  3. stop
  4. **else**
  5. send($backtrack\_msg$,$previous\_agent$)
– **stop**:
  1. send(stop, all_other_agents)
  2. done ← true

Fig. 4. SBT algorithm

other words, a backjump operation is performed from A_i to A_l. The gain in computing the inconsistent subset that includes A_l comes from avoiding the sending of $Nogood$ messages to all agents with lower priorities than A_l. A method for resolving inconsistent subsets of the $Agent\_view$, based on methods of dynamic backtrack, was presented in [Bessiere et. al.2001]. In the $DisDB$ algorithm of [Bessiere et. al.2001], agents hold explanations for every value eliminated from their domain. When a backtrack operation is performed the agent resolves its Nogood set by creating a $Nogood$ containing the union of all of its explanations, which is a set of conflicting assignments included in its $Agent\_view$. Figure 3 presents a comparison of the computational performance of ABT with full $Nogoods$ and ABT resolving $Nogoods$ by the $DisDB$ method, on a system with no message delay.

4 Synchronous Backtrak algorithms

4.1 Synchronous Backtrack (SBT)

The Synchronous Backtrack algorithm (SBT), presented in figure 4, is a distributed version of CBT [Prosser1993]. SBT has a total order among all agents, like ABT.

Agents exchange one 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, assigning its variable on the CPA and sending the CPA to the next agent. 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. If it cannot find a consistent assignment, it sends the CPA back to the previous agent to change its assignment, thus performing a chronological backtrack. An agent that receives a CPA in a backtrack message removes the assignment of its variable and tries to reassign it with a consistent value. The algorithm ends successfully if the last agent manages to find a consistent assignment for its variable. The algorithm ends unsuccessfully if the first agent encounters an empty domain.

4.2 Improvements to Synchronous search

SBT can be improved by adding to it some simple features of backjumping. As was mentioned in section 1, these features exist in ABT, independently of its asynchronicity. In the improved version of SBT, agents hold eliminating explanations for each value eliminated from their domains, as in $DisDB$. 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. As a result the synchronous algorithm gains the backjumping property. Eliminated values are returned to the domain only after a new CPA is received. Values whose eliminating $Nogoods$ are no longer consistent with the partial assignment on the CPA are returned for evaluation. This reduces the computation needed in each synchronous step.

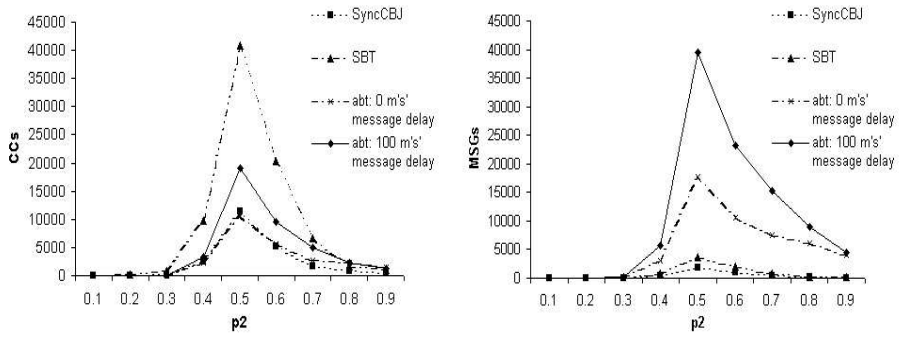


Fig. 5. (a) Number of constraints checks performed by SynCBJ SBT and ABT, (b) Total number of messages sent by SynCBJ SBT and ABT.

The resulting $Synchronous\ Conflict\ based\ Backjumping$ algorithm ($SynCBJ$) is compared in Figure 5 to SBT and ABT, with a message delay of 0 and 100 milliseconds. Figure 5(a) shows the computational effort and figure 5(b) presents the total

amount of messages sent during the algorithm run. The results presented in figure 5 show that using asynchronous search has no advantage in computational effort even on systems with no message delay. ABT is outperformed by $SynCBJ$ on systems with some message delay. In terms of message load the cost of running synchronous search is much lower than asynchronous search, as should be expected (see Figure 5(b)).

5 Discussion

Two families of search algorithms on distributed CSPs were presented, analyzed and experimentally evaluated. Asynchronous backtracking (ABT) is able to perform a type of dynamic backtrack and can resolve its Nogoods in order to store shorter versions of them. Synchronous backtracking (SBT) can use explanations in order to generate a conflict set that directs its backjumping step.

The experimental setup and the measures of performance for DisCSP algorithms were analyzed in detail. The measure of concurrent constraints checks comes closest to an implementation independent measure of computational effort in a concurrent environment. A simulator's environment, beside being essentially non concurrent, was shown to favor asynchronous algorithms by giving them the advantage of full information on the current state of the search process. An advantage that contradicts the asynchronous nature of the distributed system of a DisCSP. In order to test the concurrent performance of the two families of distributed search algorithms, a realistic environment was used. The experiments were performed on randomly generated DisCSPs, where the difficulty of the problems spans a wide range including the phase transition region. The system used message delays, either fixed or random, to test the asynchronous performance of all versions of the algorithms. Multiple versions of both families of DisCSP algorithms were run and their performance compared. The main result of all the experiments is that the best versions of ABT perform no better than the best versions of SBT, on random instances of DisCSP with randomly delayed messages.

6 Conclusions

Distributed search algorithms for solving a $DisCSP$s have two measures of performance - time, which we measure in terms of computation effort and network load [Lynch1997]. For distributed search the accepted approach is that the dominant measure is that of computational effort. This was measured standardly by the number of cycles or steps to complete the search [Lynch1997,Yokoo2000]. We measure the computational effort in terms of concurrent constraints checks ($CCCs$) [Meisels et. al.2002] in order to take in to account the computational effort required in each step. The present paper achieved three goals:

1. It introduced improved versions of synchronous distributed search, in order to compare asynchronous search to other efficient algorithms.
2. It experimented with all algorithms on a simulator that included message delays, in order to be much closer to a realistic distributed environment.
3. It used randomly generated problems with variant tightness of constraints, so that the difficulty of the problems to solve covered a wide range.

It is many times assumed that the cost in network load required for asynchronous search is compensated by gains of computational efficiency. This common assumption about distributed search on DisCSPs was found to be incorrect by our experiments reported in section 3, 4. An improved synchronous algorithm performs equally well as asynchronous search algorithms in terms of computational effort. In addition, SBT loads the network with only one message at a certain time and its performance is stable for a variety of communication qualities. Still, the idleness of the agents during most of the run of synchronous search points in the direction of concurrent search [Zivan and Meisels2002].

References

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

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

[Ginsberg1993] M. L. Ginsberg. Dynamic Backtracking. *Jof Artificial Intelligence Research.*, pages 25-46, 1993.

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

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

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

[Prosser1994] P. Prosser *Binary constraint satisfaction problems: some are harder than others* , *Proc. ECAI-94*, pp.95-99, 1994.

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

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

[Smith1994] B. M. Smith. Phase Transition and the Mushy Region in CSP. In *Proc. ECAI-94*, pages 100-104, 1994.

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

[Yokoo1992] M. Yokoo. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *12th International Conference on Distributed Computing Systems (ICDCS-92).*, pages 614-621, 1992.

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

[Yokoo2000] M. Yokoo Distributed Constraint Satisfaction Problems. Springer Verlag, 2000.

[Yokoo2000a] M. Yokoo. Algorithms for Distributed Constraint Satisfaction: A Review. *Autons Agents Multi-Agent Sys 2000*, 3(2): 198-212, 2000.

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