

# Cooperative Dynamic Multi-CBJ Search for DisCSPs\*

Uri Shapen and Amnon Meisels,  
Department of Computer Science,  
Ben-Gurion University of the Negev,  
Beer-Sheva, 84-105, Israel  
{shapenko,am}@cs.bgu.ac.il

August 13, 2007

## Abstract

A multi search process version of the sequential assignment, distributed Conflict-based BackJumping algorithm is presented. Multi-CBJ benefits greatly from sharing of *Nogoods* among search processes. It also improves when the multi-search process is dynamic, generating CBJ processes during search with a dynamic heuristic that controls load balancing. The resulting algorithm, Cooperative Dynamic Multi-CBJ, is much faster and more efficient than asynchronous algorithms like *ABT* and *AFC*.

The multi-search version of asynchronous backtracking (*ABT*) turns out to be unsuccessful. Apparently, the existing asynchronicity of *ABT* prevents it from benefiting from the use of multi search processes. Its performance actually deteriorates with additional search processes. Finally, the hypothesis that multi-search flourishes when message delays are dominant is checked. Asynchronous multi-search algorithms do perform better in the presence of message delays. However, cooperative dynamic Multi-CBJ remains the best performing multi-search algorithm, also in the presence of message delays.

## 1 Introduction

Distributed constraints satisfaction problems (*DisCSPs*) 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. [YDIK98, SGM96]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them, to

---

\*Supported by the Lynn and William Frankel center for Computer Sciences and the Paul Ivanier Center for Robotics and Production Management.

check consistency of their proposed assignments against constraints among variables that belong to different agents [Yok00, BMBM05]

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* [SGM96].

*DisCSPs* are distributed search problems, in which agents compute concurrently and independently. Consequently, the main objective for the design of search algorithms for *DisCSPs* is to achieve concurrent computation among the agents and to generate a solution in the most efficient way. Efficiency is measured by the concurrent run-time of the algorithm and by the network load that it generates [Lyn97, BMBM05, ZM06b].

Several backtracking algorithms for *DisCSPs* have been proposed in recent years, with different degrees of success at achieving concurrent efficiency. Asynchronous Backtracking (*ABT*) is an algorithm that achieves concurrency by enabling all agents to perform assignments asynchronously [YDIK98, BMBM05]. In Asynchronous Forward-Checking (*AFC*), agents assign their variables sequentially, but perform forward-checking asynchronously, achieving a high degree of concurrent computation and the resulting efficiency [MZ06]. The Concurrent Dynamic Backtracking algorithm (*ConcDB*) achieves concurrent computation by letting agents dynamically split search processes, searching concurrently non-intersecting parts of the global search space [ZM06a].

The present paper proposes a different form for achieving efficient concurrent computation for distributed search on *DisCSPs*. The proposed search algorithm uses multiple search processes that scan concurrently the complete search space. Each search process performs the sequential-assignment Conflict-based BackJumping (*CBJ*) algorithm [Pro93, BM04]. In general, Multi-Search executes several processes of sequential-assignment *DisCSP* algorithms in parallel and asynchronously. Each process starts with a different agent (first agent in its ordering) and can use any dynamic ordering heuristic. All agents participate in all search processes, assigning their variables and checking for consistency with constraining agents. Agents keep a separate data structure for each search process they participate in. In each search process agents perform assignments sequentially.

Every agent runs multiple search processes and has multiple agent views, one for each search process. Consequently, much information about different search processes is available to each agent. Multi-Search can benefit from sharing this information. The shared information can be used to select the next variable or to decide that the current assignment of another search process is inconsistent and cannot lead to a solution. This enables an agent that is currently active in a given search process to prune the search tree before receiving an explicit *Nogood* for its current search process (see section 3). Another important method for improving the efficiency of the proposed *Multi-CBJ* algorithm, is to enable a dynamic number of search processes. It is clear that the number of concurrent *SPs* must somehow help in balancing the computational load

among all agents. This can actually be achieved for the *Multi – CBJ* algorithm. It starts with a single search process and uses a simple heuristic to determine a measure of the load that is incurred by an *SP* that is currently being processed. An agent that decides that the load is too high, generates an additional search process. The dynamic number of search processes turns out to be beneficial for the efficiency of *Multi – CBJ*, as will be seen in its experimental evaluation in section 4.

Running multiple instances of asynchronous backtracking (*ABT*) was proposed in [Ham02, RH05]. The findings of [Ham02] for Multi-ABT shows a small improvement for two asynchronous backtracking search processes and a deterioration of efficiency for larger concurrency [Ham02]. To check further the usefulness of Multi-ABT, its performance was investigated on systems with message-delays and some improvement was found [RH05]. In contrast, *Multi – CBJ* as presented in the present study improves the performance of single search process, both sequential and asynchronous backtracking, by a large factor. Similar results occur for systems with random message-delays (Section 4).

Distributed constraint satisfaction problems (*DisCSPs*) are presented in section 2. Section 3 presents the principles and mechanism of the proposed multi-search algorithm, along with a detailed description of sharing, assignment timestamps and dynamic process creation. Section 4 presents an extensive experimental evaluation of *Multi – CBJ*. First, the sharing of zero-size *Nogoods* and of dynamic process generation is found to improve the run-time of *Multi – CBJ* by a large factor. Second, it is compared to *ABT* and to *AFC* and outperforms both *ABT* and *AFC*. A similar behavior is found also for systems with message delays. The discussion in section 5 presents a comparison of the proposed *Multi – CBJ* with *Multi – ABT* that runs multiple *ABT* search processes. *Multi – ABT* is slower than the cooperative and dynamic *Multi – CBJ*, both in the presence of message-delays and without it. Our conclusions are in section 6.

## 2 Distributed Constraint Satisfaction

A distributed constraint network (or a distributed constraint satisfaction problem - *DisCSP*) is composed of a set of  $k$  agents  $A_1, A_2, \dots, A_k$ . Each agent  $A_i$  contains a set of constrained variables  $X_{i_1}, X_{i_2}, \dots, X_{i_{n_i}}$ . Constraints or **relations**  $R$  are subsets of the Cartesian product of the domains of the constrained variables [Dec03]. For a set of constrained variables  $X_{i_s}, X_{j_l}, \dots, X_{m_h}$ , with domains of values for each variable  $D_{i_s}, D_{j_l}, \dots, D_{m_h}$ , the constraint is defined as  $R \subseteq D_{i_s} \times D_{j_l} \times \dots \times D_{m_h}$ . A **binary constraint**  $R_{i_j}$  between any two variables  $X_j$  and  $X_i$  is a subset of the Cartesian product of their domains;  $R_{i_j} \subseteq D_j \times D_i$ . In a distributed constraint satisfaction problem *DisCSP*, the agents are connected by constraints between variables that belong to different agents [SGM96, YDIK98]. In addition, each agent has a set of constrained variables, i.e. a local constraint network.

An assignment (or a label) is a pair  $\langle var, val \rangle$ , where  $var$  is a variable of some agent and  $val$  is a value from  $var$ 's domain that is assigned to it. A *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is a partial assignment that includes all variables of all agents, that

satisfies all the constraints. Following all former work on *DisCSPs*, 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.

One simple protocol for checking constraints, that appears in many distributed search algorithms, is to send a proposed assignment  $\langle var, val \rangle$ , of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of its variables and returns a message that either acknowledges or rejects the proposed assignment. The following assumptions are routinely made in studies of DisCSPs and are assumed to hold in the present study [BMM01, Yok00].

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

### 3 Multi-CBJ Search

Multi-Search performs several backtrack search processes (*SPs*) asynchronously on the same *DisCSP* search-space, where each *SP* is started by a different agent and runs in a different ordering of agents. Each search process includes all variables and values and therefore involves all agents and processes the whole *DisCSP* search-space. Agents in multi-search hold a set of data structures, one for each search process. These data structures, which we term Search Processes (*SPs*), include all the relevant data for the state of the agent in each of the search processes. The number of the search processes is limited and results in a linear space, in the number of processes for an agent.

Agents pass their assignments to other agents by sending messages stamped by the *ID* of the search process. Agents hold the current assignments received from other agents in the corresponding search process agent view. An agent that receives an assignment message tries to assign its local variables with values that are consistent with the assignments already on the agent view, using only the current domains in the *SP* that is related to the received search process message. Cooperation between processes is based on sharing of local information within a single agent. Information that can be shared can be *Nogoods* of other search processes, values used by other *SPs*, or the current domain of other *SPs*.

The simplest *Nogood* is of size 0, for example  $\phi \rightarrow X_i = a$ . This zero-length *Nogood* means that when  $X_i = a$  there is no solution. This type of *Nogood* can be easily shared with all search processes. No additional computation effort is needed, because a zero-length *Nogood* matches all agent view states. *Nogoods* with length  $\geq 1$

can also be shared among different *SPs*, but first one needs to check that the LHS of the *Nogood* matches the current agent view of the *SP*. This operation requests a computational effort that must be taken into consideration when evaluating the performance of the algorithm.

A sharing of a *Nogood* can lead to the ability to decide that the current partial assignment is inconsistent. This can happen even before receiving the explicit *Nogood* (e.g. a Backtracking message) for the current *SP*. The *SP* can generate a new partial assignment (e.g. a new *CPA*) and send it to the next unassigned agent. This causes several *CPA* and *Nogood/Backtracking* messages to circulate among agents simultaneously. However, there is only one *CPA* message which may potentially lead to a solution, the most updated one. The other *CPA* messages will continue to propagate and create the assignment chains down the search tree, until the *Nogood* (Backtracking) or newer messages arrive. To determine which message is the most updated and to discard obsolete messages a *time stamp* is used (cf. [NSHF04, MZ06]). Initially, all *time stamp* counters are set to zero. Each agent  $A_i$  that proposes a new assignment updates the counters as follows:

1. The counters of agents assigned before  $A_i$  are not changed.
2. The counter of  $A_i$ 's assignment is incremented by one.
3. The counters of assignment of agents located after  $A_i$  are set to zero.

An agent which receives a *time stamped* message must determine whether the received *time stamp* is more updated than its own *time stamp*. It decides by comparing the *time stamps* lexicographically. Since *time stamps* are changed according to the above rules, every two *time stamps* must have a common prefix of the assignments of agents that are *before* the current agent.

The Multi-CBJ algorithm is presented in figure 1. Each agent runs several, pre-configured, number of processes of *CBJ* search [ZM03]. A subset of the agents initializes the search by each assigning a value to its variable and sending a *CPA* message with the corresponding *ID* and *Time\_Stamp* to the next agent.

Messages exchanged by agents running *Multi - CBJ* search are the following:

1. *CPA* - a message carrying a Current Partial Assignment - list of assigned agent variables. *ID* - indicates which search process is relevant and its *Time\_Stamp*.
2. *Nogood* - a *CPA* sent in a backtrack operation, also includes *ID* and *Time\_Stamp*.
3. *stop* - a message indicating the end of the search.

Figure 1 presents the functions which are performed in any type of multi-search algorithm.

- The main function *Multi-CBJ* is run by all agents. If it is run by the agents that initialize a search process. It initializes the search by creating a new *SP* and a corresponding *CPA*. The initializing agents are configured and the **initial\_agent(id)** function returns **true** if the *ID* of the agent appears in the initial agent list. After initialization, it loops forever waiting for messages to arrive.

When receiving the message it first checks the consistency of the message by comparing the *Time\_Stamp* of the message and the current *Time\_Stamp* of the corresponding search process, by using the *Search ID* in the received message. Only if the message is consistent the current *Time\_Stamp* of the *SP* is updated and the message is taken care of by calling to the corresponding function, **assign\_CPA()** or **receive\_Nogood()**.

- **assign\_CPA** first checks if the agent holds a *SP* with the *ID* of the current *CPA* and if not, creates a new *SP*. Then it tries to find an assignment for the local variables of the agent, which is consistent with the assignments on the *CPA*. If it succeeds, it checks whether a solution needs to be reported (e.g. the *CPA* is full). Otherwise, the agent sends the *CPA* to the selected next agent. If no consistent assignment is found, the backtrack method is called.
- **receive\_Nogood** removes the current assigned value and then calls the **assign\_CPA** function to try to find a new consistent assignment. In addition, the new received *Nogood* is shared with other processes by calling the **share\_nogood()** function.
- The backtrack method is called when a consistent assignment cannot be found in a *SP*. If backtrack is called by its generator (e.g. Search ID is equal to the agent ID), the search is ended unsuccessfully. If the current agent is not the generator, a *Nogood* message is sent to the agent whose assignment is the latest of the assignments included in the inconsistent *CPA*.
- **share\_Nogood** is called when a new *Nogood* message is received. The purpose of the function is to share the new *Nogood* with other processes. It checks whether the *Nogood* matches the current state of the search process. If the states match, the nogood store is updated with the new *Nogood*. This ensures that the number of *Nogoods* in the *SP* will be limited by the number of agents times the number of values. If after the update, the current assignment is not consistent, a new assignment is generated and a new *CPA* message with an updated *Time\_Stamp* is sent to the next agent. Clearly, the use of this function must be limited to *short Nogood messages*.

The distributed *CBJ* algorithm ends successfully if the last agent finds a consistent assignment for its variables. The search fails if the first agent cannot find a consistent assignment for its variable. All Multi-*CBJ* algorithms end successfully when one of the search processes (the first one) ends successfully. The search fails if in some *SP* the leader of the search, the first agent in the assignment list, fails to find any consistent assignment.

### 3.1 Dynamic process creation

The *Multi – CBJ* algorithm has a constant number of search processes and the corresponding *CPAs* are created at the start of the algorithm's run. Search processes can be dynamically created at different points of the search by an agent generating an additional *CPA* message with a new *ID*. The need for an additional search process arises from a bad balance of load among the participating agents. This need can be detected

```

1: procedure MULTI-CBJ
2:   done  $\leftarrow$  false
3:   if initial_agent(id) then
4:     initialize_SPs
5:   end if
6:   while not done do
7:     msg  $\leftarrow$  receive_msg
8:     CPA  $\leftarrow$  msg.CPA
9:     if CPA.time_stamp  $\leq$  search current time stamp then
10:      return
11:    end if
12:    update_time_stamp(CPA.SearchID, CPA.time_stamp)
13:    switch(msg.type)
14:      stop: done  $\leftarrow$  true
15:      Nogood: receive_Nogood()
16:      CPA: assign_CPA()
17:    end while
18:    report solution
19: end procedure

1: procedure ASSIGN_CPA
2:   if first_received(CPA.SearchID) then
3:     create_SP(CPA.SearchID)
4:   end if
5:   CPA  $\leftarrow$  assign_local
6:   if is_consistent(CPA) then
7:     if is_full(CPA) then
8:       report_solution()
9:       done  $\leftarrow$  true
10:    else
11:      increment_time_stamp(CPS, id)
12:      send(CPA, next)
13:    end if
14:  else
15:    backtrack()
16:  end if
17: end procedure

1: procedure INITIALIZA_SPs
2:   CPA  $\leftarrow$  create_CPA(id)
3:   assign_CPA()
4: end procedure

1: procedure RECEIVE_NOGOOD
2:   remove_last_assignment()
3:   assign_CPA()
4:   share_nogood()
5: end procedure

1: procedure BACKTRACK
2:   if id = CPA.SearchID then
3:     CPA  $\leftarrow$  no_solution
4:     send(stop, all_other_agents)
5:     done  $\leftarrow$  true
6:   else
7:     CPA  $\leftarrow$  shortest inconsistent partial assignment
8:     backTo  $\leftarrow$  last(CPA)
9:     send(Nogood, CPA, backTo)
10:  end if
11: end procedure

1: procedure SHARE_NOGOOD
2:   for i  $\leftarrow$  1 to agents_count and i  $\neq$  id do
3:     if contains(i, CPA) then
4:       update_SP(i, CPA)
5:       if current assignment is not consistent then
6:         CPA  $\leftarrow$  get_CPA(i)
7:         assign_CPA()
8:       end if
9:     end if
10:  end for
11: end procedure

```

Figure 1: The *Multi - CBJ* algorithm

```

1: procedure CREATENEWSP
2:   if has_not_SP(id) and step_number  $\geq$  step_limit then
3:     initialize_SP()
4:   end if
5: end procedure

```

Figure 2: Dynamic process creation

by a simple heuristic that counts the average number of times that a given *CPA* was passed among the agents (without finding a solution). An alternative is to count the number of non-concurrent constraints-checks (*NCCC*) performed by the agent.

In [GS01] Gomes and Selman propose the use of random restarts during search in order to diminish the heavy tail. Their proposed method is supposed to interrupt thrashing and restart search once the effort does not seem promising anymore. In a distributed, multi-agent environment there is no need to restart the search. Dynamic search process creation exploits the distributed, multi-agent, environment to generate a new search process with a different initializing agent. Unlike centralized search, the new *SP* operates in parallel to the running search processes.

To change the code of Figure 1, so that it supports the creation of search processes dynamically during search is very simple. At the end of the **Multi-CBJ** function, the main function of the algorithm, just before **end while**, one needs to insert a call to a new function **CreateNewSP()**. The pseudo code for this function is presented in figure 2. **CreateNewSP()** first checks that there is not already a search process where the agent is a leader and whether the number of steps performed by agent has exceeded *step\_limit*. If these two conditions are met, a new process is created by calling the **initialize\_SP()** function.

## 4 Evaluating *Multi - CBJ*

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 [Lyn97, MRKZ02, ZM06b], and communication load, in the form of the total number of messages sent [Lyn97]. Non-concurrent computation effort, in systems with no message delays, are counted by a method similar to that of Lamports logical clocks [Lam78, MRKZ02, ZM06b]. Every agent holds a counter of constraint checks performed. Every message carries the value of the sending agents counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search, a measure of concurrent search effort is achieved that is close to Lamport's logical time [Lam78]. This measure can be used to count the number of non-concurrent constraint checks performed (*NCCCs*), thus incorporate the local computational effort of agents in each step [MRKZ02, ZM06b].

The experimental evaluation includes two sets of experiments. The first set in-

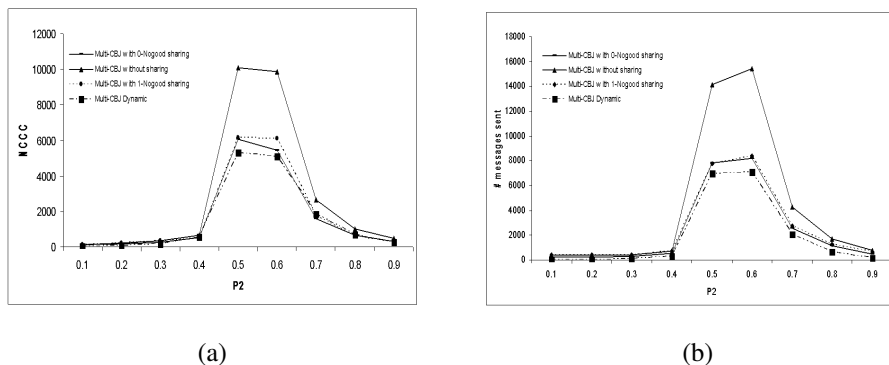


Figure 3: (a) Number of  $NCCCs$  performed and (b) Total number of message sent by  $Multi - CBJ$  with and without  $Nogood$  Sharing and Dynamic process creation on medium density  $DisCSPs$  ( $p_1 = 0.4$ )

investigates the effect of cooperation/sharing of information between search processes and of dynamic process creation on the performance of Multi-CBJ. The second set of experiments compares the cooperative dynamic Multi-CBJ algorithm to the four best performing  $DisCSP$  search algorithms. These include Synchronous Conflict Backjumping ( $SynchCBJ$ ) [ZM03], Asynchronous Forward-Checking ( $AFC$ ) [MZ06], Asynchronous Backtracking ( $ABT$ ) [Yok00, BMBM05], and Concurrent Dynamic Backtracking ( $ConcDB$ ) [ZM06a].

All experiments were conducted on an asynchronous simulator. To simulate asynchronous agents, the simulator implements agents as Java Threads. Threads (agents) run asynchronously, exchanging messages. After the algorithm is initiated, agents block on incoming message queues and become active when messages are received. Experiments were conducted on random networks of constraints. 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 [Pro96, SD96]. All sets of experiments, were conducted on networks with 15 agents ( $n = 15$ ) and 10 values for each agents variable ( $k = 10$ ). For each pair of density and tightness values ( $p_1, p_2$ ), 50 different random problems were generated and solved by each algorithm and the results presented are the average of these 50 runs.

#### 4.1 Impact of information sharing & dynamic process creation

Figure 3 presents the number of non-concurrent constraint checks ( $NCCCs$ ) performed and total messages sent by the  $Multi - CBJ$  algorithm in four different forms. Without  $Nogood$  sharing, with sharing of zero-length  $Nogoods$ , with sharing of  $Nogoods$  of length 1 and with zero-length  $Nogood$  sharing and dynamic process creation. For the dynamic process creation, the initial number of search processes is

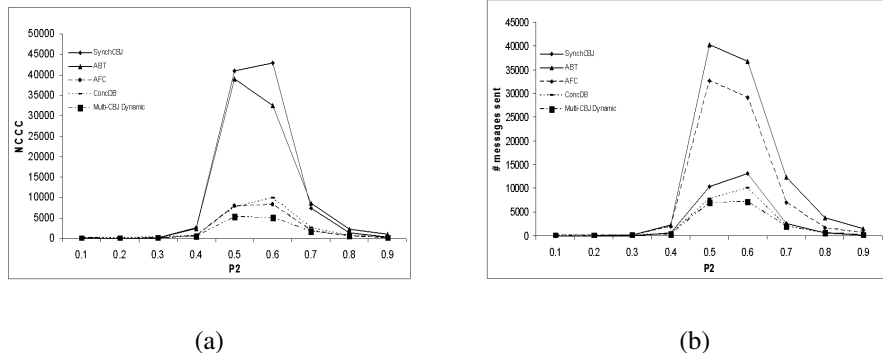


Figure 4: (a) Number of *NCCCs* performed and (b) Total number of messages sent by *Multi-CBJ*, *ConcDB*, *ABT*, *SynchCBJ* and *AFC* on medium density *DiscSPs* ( $p_1 = 0.4$ )

5 and the value of the *split\_count* parameter is 50. *split\_count* counts the average number of constraints checks performed by each process.

It is clear that sharing zero-length *Nogoods* between the *SPs* improves the runtime of *Multi-CBJ* by a large factor. In contrast, the sharing of *Nogoods* of length  $\geq 1$  produces additional computational effort and the number of *NCCCs* grows. Dynamic process creation also improves the performance of *Multi-CBJ*, in both *NCCCs* and number of messages sent. An important improvement that is the result of dynamic process creation is that improves the performance for *easy problems* (e.g. for small values of  $p_2$ ). The resulting performance equals that of *SynchCBJ* and is a bit hard to see on Figure 3.

## 4.2 Comparing to other *DiscSP* algorithms

In order to evaluate the performance of *Multi-CBJ* it is compared to the four best known *DiscSP* search algorithms. Synchronous Conflict Backjumping (*SynchCBJ*) is a distributed version of the well known *CBJ* algorithm [Pro93, BM04]. Asynchronous Forward-checking (*AFC*) [MZ06] is an algorithm which performs sequential assignments like *CBJ*, but distributes the computation of forward-checking asynchronously among all agents. As a result, *AFC* was found to have a very good concurrent performance and to outperform *ABT* by a large margin [MZ06].

Asynchronous Backtracking (*ABT*) [BMBM05, Yok00] is the classical distributed search algorithm on *DiscSPs*. 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. In the present implementation of *ABT*, *Nogoods* are resolved and stored according to the method presented in [BMBM05]. Based on

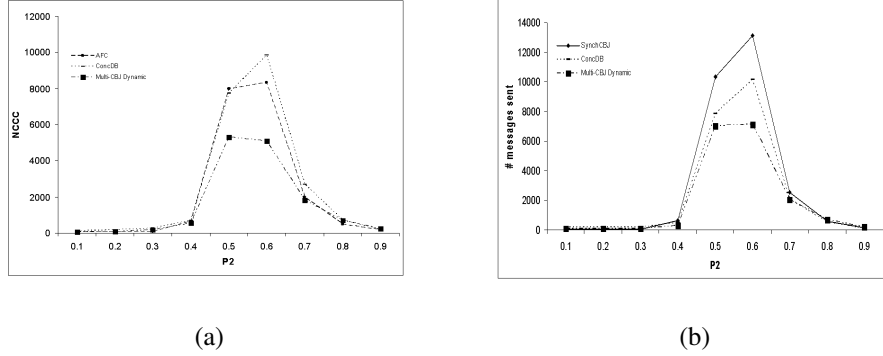


Figure 5: A closer look at figure 4

Yokoos suggestions [Yok00], agents read in every step all messages received before performing the computation. This forms the best performing version of *ABT*. Concurrent Dynamic Backtracking (*ConcDB*) [ZM06a] is another asynchronous backtracking algorithm that utilizes multiple concurrent search processes on *disjoint parts* of the *DisCSP* search-space.

All four algorithms are compared to the best version of the *Multi-GBJ* algorithm (see section 4.1), which uses both sharing of zero-size *Nogoods* and dynamic process creation. Figure 4 presents the number of non-concurrent constraint checks performed and the total number of messages sent by *Multi-GBJ*, *ConcDB*, *SynchCBJ*, *AFC* and *ABT* on problems with medium constraint density ( $p_1 = 0.4$ ). *Multi-GBJ* outperforms *ABT* and *SynchCBJ* by a large factor and even sends less messages than (the *single sequential process*) *SynchCBJ*. In order to observe better the advantage of *Multi-GBJ* over its competitors one can take off the two worst performing algorithms from each of the figures in 4. Figure 5 presents a closeup view of each of the performance measures. Each figure includes just the three best performing algorithms in that measure. *Multi-GBJ* is the *best performing algorithm in both measures*. It performs less *NCCCs* than the two fastest *DisCSP* search algorithms - *AFC* and *ConcDB*. It sends less messages than *SynchCBJ* and *ConcDB*. The improvement in the number of messages sent of the *Multi-GBJ* over *AFC* is significant and can even be seen in figure 4. *Multi-GBJ* improves the best known algorithms by about 30%.

## 5 Discussion

The *Multi-GBJ* algorithm was shown to be the best performing *DisCSP* algorithm. When multiple sequential assignment search processes (e.g. *CBJ*) are combined, share *Nogoods*, and search processes are dynamically generated, the resulting search algorithm runs faster than the best algorithms and sends less messages. An immediate question comes into mind. What would happen if a multi-search algorithm

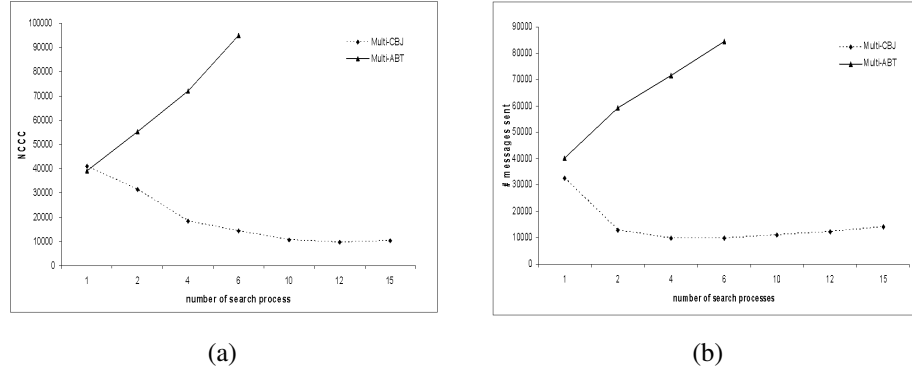


Figure 6: (a) Number of *NCCCs* performed and (b) Total number of message sent by *Multi - CBJ* and *Multi - ABT* on hard instances of *DisCSPs* ( $p_1 = 0.4, p_2 = 0.5$ ), against number of search processes

would use as its building block an asynchronous assignment search algorithm, such as *ABT*. This was attempted in the past by Hamadi et. al [Ham02]. The results of [Ham02] were not very encouraging for number of processes larger than 2.

Figure 6 presents the performance of *Multi - CBJ* and *Multi - ABT* against the number of search processes used. The two standard measures of performance are depicted - number of *NCCCs* performed and the total number of messages sent. Both *Multi - CBJ* and *Multi - ABT* were run on the hardest instances of *DisCSPs* ( $p_1 = 0.4; p_2 = 0.5$ ). *Multi - CBJ* improves its performance when the number of the concurrent search processes grows. In contrast, *Multi - ABT* performs more *NCCCs* and sends more messages, as the number of search processes grows. One can say that the performance of *ABT* deteriorates with the multi-search approach, which was already observed by [Ham02].

Recently, Ringwelski and Hamadi have proposed the use of *Multi - ABT* for systems with message delays [RH05]. Figure 7 presents the performance of *Multi - CBJ* and *Multi - ABT* against the number of search processes used for systems with random message delays. Both *Multi - CBJ* and *Multi - ABT* were run on the hardest instances of *DisCSPs* ( $p_1 = 0.4; p_2 = 0.5$ ), as in the case of no delays in figure 6. The interesting result is that *Multi - CBJ* improves its run-time (*NCCCs*) when the number of concurrent search processes grows but sends a little more messages. However, *Multi - CBJ* performs better than *Multi - ABT* when the number of concurrent processes is larger than 8 (for our experimental set-up with 15 agents). For the network load performance measure, *Multi - ABT* is extremely worse than *Multi - CBJ*. Note that the version of *Multi - CBJ* that is used in these experiments has a *fixed number of search processes* and therefore is not the best version of the algorithm. It also does not share *Nogoods*.

The advantage of Multi-Search for both sequential-assignment and asynchronous-assignment algorithms (*Multi - CBJ* and *Multi - ABT*) is more pronounced in the

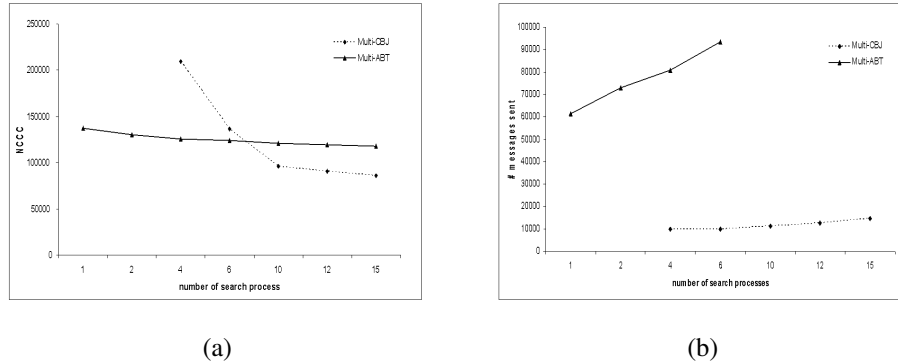


Figure 7: (a) Number of *NCCCs* performed and (b) Total number of message sent by *Multi-CBJ* and *Multi-ABT* on hard instances *DisCSPs* problems ( $p_1 = 0.4$ ,  $p_2 = 0.5$ ) on the systems with message delays

presence of message delay. This may be related to some general properties of search by concurrent processes. Message delays tend to influence only some of the running search processes. Therefore, the probability to delay the global search process becomes smaller as the number of search processes grows [ZM06b]. This agrees in general with the results for message delays in [RH05]. This makes the last result even more striking. Even in systems with message delays, the *Multi-CBJ* algorithm is more efficient (in both performance measures), than *Multi-ABT*.

## 6 Conclusion

A cooperative Dynamic Multi Search algorithm for distributed CSPs has been presented. The algorithm runs multiple instances of the sequential-assignment, distributed, Conflict-based BackJumping (*CBJ*) algorithm. *Nogood* sharing among all search processes enables an early termination of search processes which do not lead to a solution. Search processes are dynamically generated by agents in an asynchronous distributed process. Cooperative Dynamic *Multi-CBJ* provides an efficient method for multiple search processes to search concurrently a *DisCSP*. Dynamic process generation establishes a method for load balancing among all participating agents. For the more difficult problems more search processes are dynamically generated.

An extensive experimental evaluation of *Multi-CBJ* has also been presented. Its experimental behavior on randomly generated *DisCSPs* clearly indicates its efficiency, compared to concurrent algorithms of a single search process like *ABT*, *AFC* and multi search algorithms like *ConcDB* and *Multi-ABT*. Finally, a major advantage of the proposed *Multi-CBJ* algorithm is its simplicity. Unlike asynchronous-assignment algorithms it does not need idle detection to find a solution [BMBM05]. Its basic search process is the well-known *CBJ* algorithm and it can easily be enhanced to use ordering heuristics [BM04].

## References

- [BM04] I. Brito and P. Meseguer. Synchronous, asynchronous and hybrid algorithms for discsp. In *Workshop on Distributed Constraints Reasoning(DCR-04) CP-2004*, Toronto, September 2004.
- [BMBM05] C. Bessiere, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161:1-2:7–24, January 2005.
- [BMM01] C. Bessiere, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *Proc. Workshop on Distributed Constraints (in IJCAI-01)*, 2001.
- [Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufman, 2003.
- [GS01] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126:43–62, 2001.
- [Ham02] Y. Hamadi. Interleaved backtracking in distributed constraint networks. *Intern. Jou. AI Tools*, 11:167–188, 2002.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, 2:95–114, April 1978.
- [Lyn97] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.
- [MRKZ02] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.
- [MZ06] A. Meisels and R. Zivan. Asynchronous forward-checking for distributed csp. *Constraints*, 16:132–156, 2006.
- [NSHF04] T. Nguyen, D. Sam-Hroud, and B. Faltings. Dynamic distributed backjumping. In *Proc. 5th workshop on distributed constraints reasoning DCR-04*, pages 46–61, Toronto, September 2004.
- [Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
- [Pro96] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
- [RH05] G. Ringwelski and Y. Hamadi. Boosting distributed constraint satisfaction. In *Proc. Constraints Processing 2005 (CP-05)*, pages 549–562, Sitges, Spain, 2005.
- [SD96] B. M. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

- [SGM96] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsp). In *Constraint Processing-96, (short paper)*, pages 561–2, Cambridge, Massachusetts, USA, October 1996.
- [YDIK98] 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.
- [Yok00] M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.
- [ZM03] R. Zivan and A. Meisels. Synchronous vs asynchronous search on discsp. In *Proc. 1st European Workshop on Multi Agent System, EUMAS*, Oxford, December 2003.
- [ZM06a] R. Zivan and A. Meisels. Concurrent search for distributed csps. *Artificial Intelligence*, 170:440–461, 2006.
- [ZM06b] R. Zivan and A. Meisels. Message delay and discsp search algorithms. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 46:415–439, 2006.