Concurrent Search for Distributed CSPs[*]

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

Abstract

A distributed concurrent search algorithm for distributed constraint satisfaction problems (*DisCSPs*) is presented. Concurrent search algorithms are composed of multiple search processes (*SPs*) that operate concurrently and scan non-intersecting parts of the global search space. Each *SP* is represented by a unique data structure, containing a current partial assignment (CPA), that is circulated among the different agents. Search processes are generated *dynamically*, started by the initializing agent, and by any number of agents during search.

In the proposed, $ConcDB$, algorithm, all search processes perform *dynamic backtracking*. As a consequence of backjumping, a search space can be found unsolvable by a different search process. This enhances the efficiency of the $ConcDB$ algorithm. Concurrent Dynamic Backtracking is an asynchronous distributed algorithm and is shown to be faster than former algorithms for solving $DisCSPs$. Experimental evaluation of $ConcDB$, on randomly generated $DisCSPs$ demonstrates that the network load of $ConcDB$ is similar to the network load of synchronous backtracking and is much lower than that of asynchronous backtracking. The advantage of *Concurrent Search* is more pronounced in the presence of imperfect communication, when messages are randomly delayed.

Key Words: Constraints Satisfaction, Search, Distributed AI.

1. Introduction

Distributed constraint satisfaction problems (*DisCSPs*) are composed of agents, each holding its local constraint 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. [20, 19]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages with other agents, to check consistency of their proposed assignments against constraints with variables owned by different agents. [20, 1].

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*

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

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 [19].

A search procedure for a consistent assignment of all agents in a $DisCSP$, is a distributed algorithm. 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.

One method for achieving concurrency for search on $DisCSPs$ is for agents to perform assignments concurrently. 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 for information about all relevant assignments of higher priority agents [21, 16, 1]. 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 [20, 1].

The present paper proposes a different way of achieving concurrency for search. In order to achieve shorter overall run-time, concurrent search runs multiple search processes on a *DisCSP*. 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. In each search process agents perform assignments sequentially. Agents and variables are ordered randomly on each of the search processes, diversifying the sampling of the search space. Concurrent search distributes among agents a dynamically changing number of search processes. The degree of concurrency during search changes dynamically and enables automatic load balancing.

In order to exchange complete information about assignments of all agents, *Concurrent Search* circulates multiple *Current Partial Assignments*(*CPA*s) among all agents. Each *CPA* represents one search process (SP) and each search process scans a different part of the global search space. The search space is split dynamically at different points on the path of the search process by agents generating additional CPAs. The splitting and re-splitting of the search space is performed independently by agents and is thus a distributed process. Splits are first attempted by agents residing at the top of the sub search tree that is being split. When the agent has no opportunity to split, the splits move down to agents that are lower in the split sub-tree.

It is well known that search algorithms on CSPs benefit from backjumping [14, 6, 4]. An elegant form of backjumping that can be easily adapted to distributed CSP algorithms, is *Dynamic Backtracking* [7]. The best Concurrent Search algorithm, Concurrent Dynamic Backtracking ($ConcDB$), presented in this paper, performs dynamic backtracking (DB) on each of its concurrent sub-search spaces. 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 that are discovered to be unsolvable. The combination of *Concurrent Search* and *Dynamic Backtracking* in $ConcDB$ result in a fast algorithm, which outperforms previous complete search algorithms both sequential assigning (synchronous) and Asynchronous Backtracking by a large factor (See Section 5).

Concurrent Search, as proposed in the present paper and in [22, 24], may seem similar to former approaches of parallelism. Splitting the search space at the first agent and running several search processes for each of the values of the first agents' domain is part of interleaved search in [8]. The possibility of dynamic allocation of predefined search slots for multiple asynchronous backtracking search processes was reported in [17]. The performance of multiple asynchronous backtracking search shows a small improvement for two asynchronous backtracking search processes and deteriorates for larger concurrency [8]. On the other hand, *Concurrent Search* as presented in the present study improves the performance of single search process, both sequential and asynchronous backtracking, by a large factor. (see Section 5).

Distributed constraint satisfaction problems ($DisCSPs$) are presented in section 2. Section 3 presents the principles and mechanism of *Concurrent Search* along with a detailed description of Concurrent Dynamic Backtracking. The method for terminating SPs which are shown to be unsolvable during backjumps, is presented in section 3.3. A correctness and completeness proof for *Concurrent Search*, and in particular for $ConcDB$, is presented in section 4.

Section 5 presents an extensive experimental evaluation, which demonstrates multiple advantages of $ConcDB$. First, the features of concurrent search are investigated. Multiple search processes are better than one and dynamic generation of search processes makes the algorithm perform best (section 5.1). Next, a comparison of $ConcDB$ to existing algorithms for solving $DisCSPs$, is presented in section 5.2. The impact of message delay on the performance of concurrent search is evaluated in Section 5.3. $ConcDB$ is compared to CBJ, AFC and ABT in the presence of random message delays. Sequential assigning algorithms are affected the most by message delays. Concurrent dynamic backtracking is much more robust to message delay than asynchronous backtracking. The delay of messages has a strong impact on the run of distributed search algorithms [5, 26]. Finally, the heuristic which is used by agents to determine the level of concurrency is evaluated in Section 5.4. This experiment demonstrates the size of the effect that the level of concurrency has on the performance of the algorithm in the presence of message delay. Conclusions on the advantages of using multiple DB search processes in $DisCSP$ search are presented 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 $A_1, A_2, ..., A_k$. Each agent A_i contains a set of constrained variables $X_{i_1}, X_{i_2}, ..., X_{i_{n_i}}$. Constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables [4]. For a set of constrained variables $X_{i_s}, X_{j_l}, ..., X_{m_h}$, with domains of values for each variable $D_{i_s}, D_{j_l}, ..., D_{m_h}$, the constraint is defined as $R \subseteq D_{i_s} \times D_{j_l} \times ... \times D_{m_h}$. 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. [21, 19]). In addition, each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair $< var, val >$, where var is a variable of some agent and val is a value from var's domain that is assigned to it. A *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is a partial assignment that includes all variables of all agents, that satisfies all the constraints. Following all former work on *DisCSP*s, agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. An agent can send messages to any one of the other agents.

One simple protocol for checking constraints, that appears in many distributed search algorithms, is to send a proposed assignment $< var, val >$, of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of its variables and returns a message that either acknowledges or rejects the proposed assignment. The following assumptions are routinely made in studies of $DisCSP$s and are assumed to hold in the present study [20, 2].

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

3. Concurrent Search

Concurrent Search is a family of algorithms which perform multiple concurrent backtrack search processes asynchronously on disjoint parts of the *DisCSP* search-space. Each search process includes all variables and therefore involves all agents. Each agent holds 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 on each of the search processes. Agents in concurrent search algorithms pass their assignments to other agents on a special type of message - a Current Partial Assignment (CPA). Each *CPA* represents one search process, and holds the agents' current assignments in the corresponding search process. An agent that receives a *CPA* tries to assign its local variables with values that are not conflicting with the assignments already on the *CPA*, using only the current domains in the *SP* that is related to the received *CPA*. The uniqueness of the *CPA* for every search space ensures that assignments are not done concurrently (and conflictingly) in a single sub-search-space [24, 25].

An agent can generate a set of SPs and corresponding CPAs that split the search space of a single SP whose CPA has 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 section 3.2). *CPA*s are created either by the Initializing Agent (*IA*) at the beginning of the algorithm run, or dynamically by any agent that splits an active search-space during the algorithm run. A simple heuristic of counting the number of times agents pass a given *CPA* (without finding a solution), is used to determine the need for re-splitting of the search-space traversed by that CPA. This generates a mechanism of load balancing, creating more search processes on heavily backtracked search spaces.

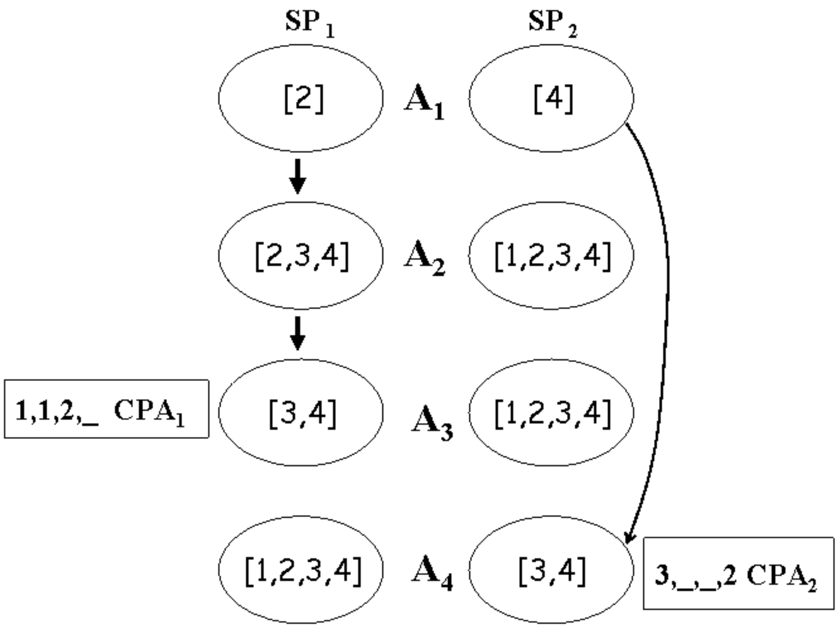

Figure 1: Simple *Concurrent Search* with two CPAs

Figure 1 presents an example of a DisCSP, searched concurrently by two search processes represented by two CPAs, CPA_1 and CPA_2. Each of the four agents A_1 to A_4, holds two Search Processes (SPs). The domains of all four agents are the same - $\{1..4\}$. The current domains of the SPs are shown in Figure 1. The domains on the left represent the state after 3 assignments to CPA_1. The domains on the right hand side of figure 1 represent the state after the second assignment to CPA_2.

Agent A_1 has assigned the value 1 on CPA_1 and the value 3 on CPA_2. The values that are left in each of its domains are 2 in SP_1 and 4 in SP_2. Agent A_3 has assigned the value 2 to CPA_1, having failed to assign the value 1. This leaves its current domain, for SP_1, with the values [3,4]. The two CPAs are traversing non intersecting sub search spaces in which CPA_1 is exploring all tuples beginning with 1 or 2 for agent A_1, and CPA_2 all tuples beginning with 3 or 4. CPA_1 is depicted on the LHS of figure 1 and CPA_2 is on the RHS. CPA_1 moves among the agents in the order $A_1 \rightarrow A_2 \rightarrow A_3$. CPA_2 moves in the order $A_1 \rightarrow A_4 \rightarrow$...

A backtrack operation is performed by an agent which fails to find a consistent assignment with the partial assignment on the *CPA* that it is currently holding. A backtrack operation sends a CPA backwards, requesting the receiving agent to revise its assignment on the CPA. Agents that have performed dynamic splitting, have to collect all of the returning CPAs, of the relevant SP, before declaring that a sub-search-space does not contain a solution. In this case all consistent values of the split domain have been sent forward on some CPA and failed, which means the agent must perform a backtrack operation.

The search ends unsuccessfully, when all *CPA*s return for backtrack to the IA and the domain of the first variable of each CPA is empty. In this case all the search processes are stopped. The search ends successfully if *one CPA contains a complete assignment*, a value for every variable in the $DisCSP$.

There is no synchronization between the assignments performed in different SPs and the splitting of different CPAs. Due to the random choice of the next agent and the dynamic asynchronous splitting of search spaces, the steps of agents in different search process are interleaved in a non predefined order. This makes Concurrent Search algorithms asynchronous [11].

The following subsections present a detailed description of Concurrent Search.

3.1 Main objects of Concurrent search

The main data structure that is used and passed between the agents is a *current partial assignment (CPA)*. A *CPA* contains an ordered list of triplets $< A_i, X_j, val >$ where A_i is the agent that owns the variable X_j and val is a value, from the domain of X_j, assigned to X_j. This list of triplets starts empty, with the agent that initializes the search process, and includes more assignments as it is passed among the agents. Each agent adds to a *CPA* that passes through it, a set of assignments to its local variables that is consistent with all former assignments on the *CPA*. If successful, it passes the *CPA* to the next agent. If not, it *backtracks*, by sending the *CPA* to the agent from which it was received.

Splitting the search space on some variable divides the values in the domain of this variable into several groups. Each sub-domain defines a unique sub-search-space and a unique *CPA* traverses this search space. Dynamic splitting is triggered by the number of assignment steps performed on a CPA, without returning back to its initiator. This is an intuitive meaning of thrashing and can be based on a simple threshold for the number of unsuccessful assignments - $steps\_limit$.

Consider the constraint network that is described in figure 2. The 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 2. In order to keep the example small, no initial split is performed, only dynamic splitting. The value of $steps\_limit$ in this example is 4. The first 5 steps of the algorithm run produce the state that is depicted on the RHS of Figure 2. The circled values in the current domains of agents X_1 and X_2 are the assigned values on the CPA. The current domain of X_2 had only two values left, $[3, 4]$. X_3 is now holding the CPA and has no assignment that is consistent with it.

The run of the algorithm during these 5 steps is as follows:

1. X_1 assigns its variable the value 1, and sends to X_2 a CPA with a step counter CPA_steps = 1.

2. X_2 assigns its variable the value 2, and sends the CPA with both assignments, and 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 reassigns its variable with the value 3, and sends the CPA again to X_3 after raising the step counter to 4.

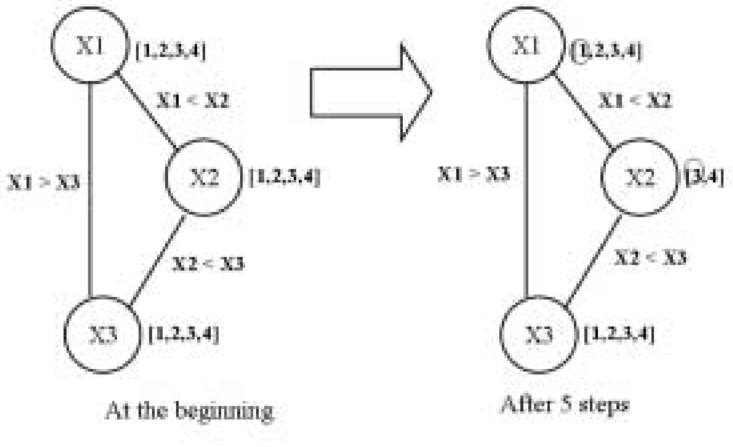

Figure 2: Initial state and the state after the CPA travels 5 steps without returning to its initializing agent

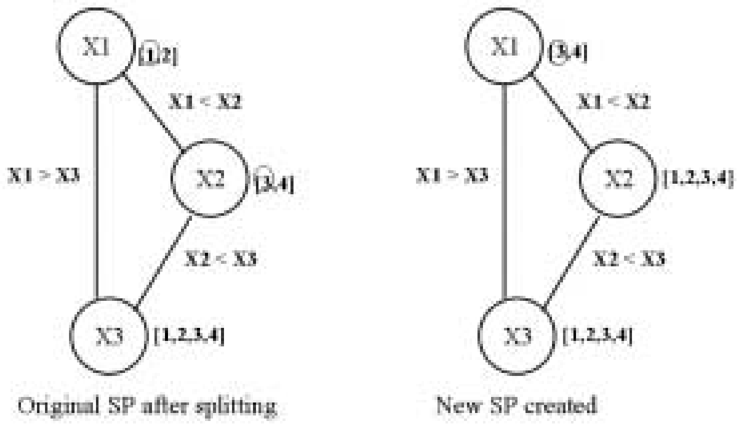

Figure 3: The new non intersecting search spaces now searched using two different *CPAs*

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 $step\_limit$. It has to generate a split operation. Before trying to find an assignment for its variable, X_3 sends a split message to X_1 which is the $CPAs$ generator and changes the value of the CPA_steps counter to 0. Next, it sends the CPA to X_2 in a backtrack message. The algorithm run proceeds as follows:

- When X_1 receives the split message it performs the following operations:

  - Creates a new (empty domain) SP data structure.
  - Deletes values 3 and 4 from its original domain and inserts them into the domain of the new split SP.
  - Creates a new CPA and assigns it with 3 (a value from the new domain).
  - Sends the new CPA to a randomly selected agent.

- Other agents that receive the new CPA create new SPs with a copy of the initial domain.

The resulting split search-spaces are depicted in Figure 3. Circled values represent those that are currently assigned on the corresponding CPA_1 or CPA_2. 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 3). 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 current domain of X_2 (on SP_1) does not contain values 1 and 2 which were eliminated in earlier steps. 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 3).

Every agent that receives a *CPA* for the first time, creates a local data structure which we call a *search process (SP)*. This is true also for the initializing agent (*IA*), for each created *CPA*. The *SP* holds all data on current domains for the variables of the agent, such as the remaining and removed values during the path of the *CPA*.

The structure of the *ID* of a *CPA* and its corresponding *SP* is a pair $< A, j >$, where A is the ID of the agent that created the CPA and j is the number of CPAs this agent created so far. This enables all agents to create *CPA*s with a unique *ID*. When a split is performed during search, the generated CPA has a unique ID and carries the ID of the CPA from which it was split.

Although any agent can split its domain, the current version of the algorithm splits search spaces as high as possible in the search tree. This generates split sub-search-spaces that are as large as possible and a larger number of agents participate in the divided search procedure. When agents have no further opportunity to split an SP because of lack of values in their current domain, split messages are transferred down the search tree to agents lower in the current order of the search (See Figure 5, procedure $perform\_split$, lines 2, 8 and 9). Note that different concurrent search processes are ordered differently. Therefore, the splitting of the search space occurs at different agents in different concurrent SPs.

3.2 General Concurrent Search

The following terminology is used in the description of concurrent search algorithms:

- $CPA\_generator$: Every *CPA* carries the *ID* of the agent that created it.
- $origin\_SP$: an agent that performs a dynamic split, holds in each of the new *SPs* the *ID* of the *SP* it was split from (i.e. of $origin\_SP$). An analogous definition holds for $origin\_CPA$. The $origin\_SP$ of an *SP* that was not created in a dynamic split operation is its own *ID*.

```
Concurrent_Search:
1.  done ← false
2.  if(IA) then initialize_SPs
3.  while(not done)
4.     switch msg.type
5.        $split$: perform_split
6.        $stop$: done ← true
7.        $CPA$: receive_CPA
8.        $backtrack$: receive_CPA

receive_CPA:
1.  CPA ← $msg.CPA$
2.  if(first_received(CPA.ID))
3.     create_SP(CPA.ID)
4.  if(CPA.generator = ID)
5.     CPA.steps ← 0
6.  else
7.     CPA.steps ++
8.     if(CPA.steps = $steps\_limit$)
9.        $splitter \leftarrow select\_assigned\_agent$
10.       $CPA\_steps \leftarrow 0$
11.       send($split\_msg$ $splitter$)
12. if(msg.type = $backtrack$)
13.    remove_last_assignment
14. $assign\_CPA$

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

initialize_SPs:
1.  for i ← 1 to $domain\_size$
2.     $CPA$ ← create_CPA(i)
3.     SP[i].domain ← first_var[value_i]
4.     create_SP(CPA.ID)
5.     $assign\_CPA$
```

Figure 4: Main and Assign parts of Concurrent Search

- $split\_set$: the set of SP IDs, stored in an $origin\_SP$. Every $origin\_SP$ holds in its $split\_set$ the IDs of all the SPs for which it is their $origin$ (i.e. all SPs which were split from it by the agent holding it). For every active SP, the only $split\_set$ relevant is the $split\_set$ of its $origin\_SP$.
- $steps\_limit$: the number of steps (from one agent to the next) that will trigger a split, if the *CPA* does not find a solution, or does not return to its generator.

The messages exchanged by agents in concurrent search are the following:

- **CPA** - the message carrying a *Current Partial Assignment*.
- **backtrack_msg** - a CPA sent in a backtrack operation.
- **stop** - a message indicating the end of the search.

```
backtrack:
1.  delete(CPA.ID from $origin\_SP.split\_set$)
2.  if($origin\_SP.split\_set$ is_empty)
3.      if(IA)
4.          CPA ← no_solution
5.          if(no_active_CPAs)
6.              report_no_solution
7.              stop
8.      else
9.          send($backtrack$, $last\_assignee$)
10. else
11.     $mark\_fail$(CPA)
```

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

```
stop:
1.  send($stop$, $all\_other\_agents$)
2.  done ← true
```

Figure 5: Backtrack and Split for Concurrent Search

- **split** - a message that is sent in order to trigger a split operation. Contains the *ID* of the *SP* to be split.

Figures 4 and 5 present the functions which are performed in any type of concurrent search algorithm. The main function of the algorithm and functions that perform assignments on the CPA when it moves forward are presented in Figure 4.

- The main function **Concurrent_Search** is run by all agents. If it is run by the *initializing agent (IA)*, it initializes the search by creating multiple *SPs*, assigning each SP with one of the first variable's values. After initialization, it loops forever, waiting for messages to arrive.

- **receive_CPA** first checks if the agent holds a *SP* with the *ID* of the *current_CPA* and if not, creates a new *SP*. If the CPA is received by its generator, it changes the value of the steps counter (CPA_steps) to zero. This prevents unnecessary splitting. Otherwise, it checks whether the *CPA* has reached the $steps\_limit$ and a split must be initialized (lines 7-9). The splitting agent, which we term $splitter$, is selected to be any one of the assigned agents (line 9). A specific heuristic for splitting is to send the split message to the $CPA\_generator$ (as mentioned above the $CPA\_generator$ is the first part of any CPA_ID). This is equivalent to splitting the search tree as high as possible. This specific policy is implemented in the $ConcDB$ version of the present paper. Before assigning the CPA a check is made whether the CPA was received in a $backtrack\_msg$. If so, the previous assignment of the agent which is the last assignment made on the CPA is removed, before $assign\_CPA$ is called (lines 12-13).

- **assign_CPA** tries to find an assignment for the local variables of the agent, which is consistent with the assignments on the CPA. If it succeeds, the agent sends the *CPA* to the selected $next\_agent$ (line 7). If not, it calls the *backtrack* method (line 9).

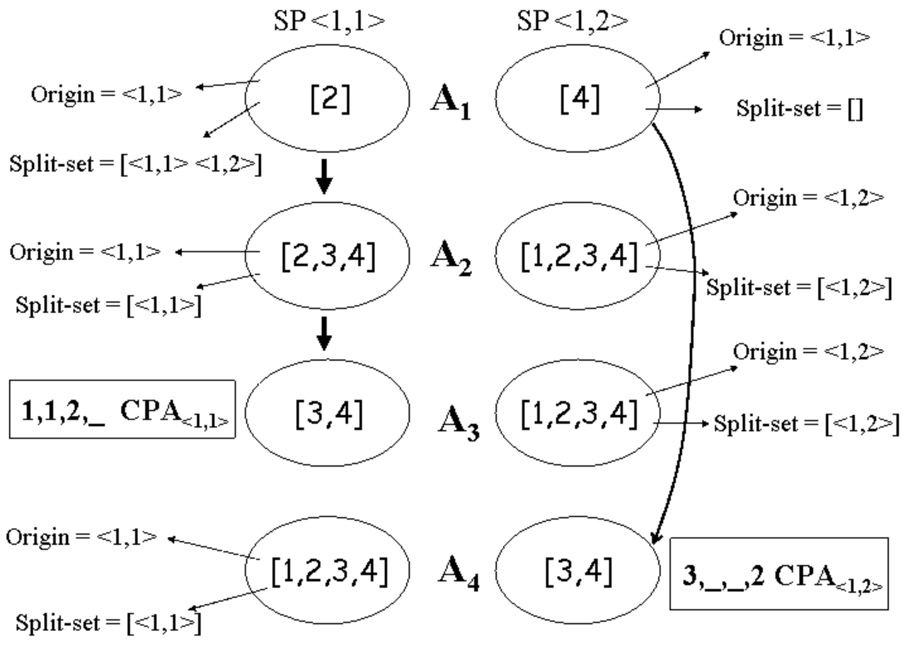

Figure 6: Simple *Concurrent Search* with two CPAs

The rest of the functions of Concurrent Search are presented in Figure 5.

- The **backtrack** method is called when a consistent assignment cannot be found in a *SP*. Since a split might have been performed by the current agent, a check is made, whether all the *CPA*s in the $split\_set$ of the $origin\_CPA$ of the backtracking CPA have also failed (line 2). If not then only the current CPA is marked (line11) and no further action need take place. When all split CPAs have returned unsuccessfully, the search space of the SP is unsolvable and a backtrack operation is initialized.

  In case of an *IA*, the *SP* and the corresponding *origin_CPA* are marked as a failure (lines 3-4). If all other *CPA*s are marked as failures, the search is ended unsuccessfully (line 6). If the current agent is not the IA, a backtrack message is sent to the agent whose assignment is the latest of the assignments included in the inconsistent CPA (line 9).

- The **perform_split** method tries to find in the *SP* specified in the $split\_message$, a variable with a non-empty current_domain. It first checks that the CPA to be split has not been sent back already, in a backtrack message (line 1). If it does not find a variable for splitting, it sends a split_message to $next\_agent$ (lines 8-9). If it finds a variable to split, it creates a new *SP* and CPA, and calls *assign_CPA* to initialize the new search (lines 3-5). The *ID* of the generated CPA is added to the split set of the divided SPs $origin\_SP$ (line 6).

Figure 6 extends the example presented in figure 1. For each SP (except for the ones holding the corresponding CPA), the content of the $origin$ and the *split-set* are displayed. The $origin$ of all SPs except

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

receive_CPA:
1.  CPA $\leftarrow msg.CPA$
2.  if(unsolvable SP)
3.    terminate CPA
4.  else
..
..
..
11.   if($CPA\_steps$ = $steps\_limit$)
12.     $splitter \leftarrow CPA\_generator$
13.     send($split\_msg$, $splitter$)
14.   if(msg.type = $backtrack$)
15.     check_SPs(CPA.inconsistent_assignment)
16.     last_sent_CPA.$remove\_last\_assignment$
17.     $CPA \leftarrow last\_sent\_CPA$
18.   if($sp$.split_ahead)
19.     send(unsolvable, $sp$.next_agent)
20.     $sp$rename_SP
21. $assign\_CPA$

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

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

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

Figure 7: Methods for Dynamic Backtracking of $ConcDB$

for the SPs of agent A_1 are their own IDs since they were not created in a dynamic split operation. Their *split-set* includes only their own ID, since they are not yet an origin of any SP created by a dynamic split operation. In this example the $SP <1,2>$ held by agents A_2 and A_3 will only be created when they will first receive the corresponding CPA.

The $origin$ of $SP <1,2>$ is the SP it was split from which is $<1,1>$. The split set of $SP <1,2>$ is empty since the relevant *split-set* is only its $origin\_SP$'s *split-set*. The *split-set* of $SP <1,1>$, includes its own ID and the ID of the SP that was split from it which is $<1,2>$

3.3 Concurrent Dynamic Backtracking

The method of backjumping that is used in the $ConcDB$ algorithm is based on *Dynamic Backtracking* [7]. Each agent that removes a value from its current domain stores the partial assignment that caused the removal of the value. This stored partial assignment is called an *eliminating explanation* by [7]. When the current domain of an agent empties, the agent constructs a backtrack message from the union of all assignments in its stored removal explanations. The union of all removal explanations is an inconsistent partial assignment, or a *Nogood* [7, 20]. The backtrack message is sent to the agent which is the owner of the most recently assigned variable in the inconsistent partial assignment.

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

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

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

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

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

Figure 7 presents the methods ***ConcDB***, ***receive_CPA*** and ***backtrack***, that were changed from the general description of *Concurrent Search* in Figures 4 and 5. Figure 7 contains also two additional methods needed for adding *Dynamic Backtracking* to concurrent search.

In method $receive\_CPA$ a check is made in lines 2,3 whether the SP related to the received CPA is marked unsolvable. In such a case the CPA is not assigned and the related SP is terminated. If the

$split\_limit$ is reached the $split$ message is sent to the generator of the CPA to create the split as high as possible in the search tree (lines 11-13). This is a specific heuristic for $select\_assigned\_agent$ of the general $receive\_CPA$ in Figure 4 (line 9 there). For a backtracking CPA (lines 14-20) a check is made whether there are other SPs which can be declared unsolvable. This can happen when the head (or prefix) of their partial assignment (their *common head i.e. CH*) contains the received inconsistent partial assignment. Procedure $check\_SPs$ for every such SP found, initiates the termination of the search process on the unsolvable sub-search-space and resumes the search with a newly generated CPA. Next, a check is made whether the SP was split by agents who received the CPA after this agent (line 18) (this fact can be recorded on the CPA when its holder initiates the split). If so, the termination of the unsolvable SP is initiated by sending an $unsolvable$ message. A new ID is assigned to the received CPA and to its related SP (line 20).

The inconsistent partial assignment received in the $backtrack$ message may rule out more than one active search process. The check performed by the function ***check_SPs*** triggers the termination of these inconsistent search processes. For each of the terminated SPs a new CPA is created and the search process is resumed after the culprit assignment is revised.

In method $backtrack$, the agent inserts the culprit inconsistent partial assignment into the backtrack message (line 9) before sending it back in line 10. This is the only difference from the standard backtrack method in Figure 5.

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

4. Correctness of Concurrent Search

To prove correctness of a search algorithm for $DisCSPs$ one needs to prove that it is sound, complete and that it terminates. A central fact that can be established immediately is that agents send forward only consistent partial assignments. This fact can be seen at lines 1, 2 and 7 of procedure *assign_CPA* (Figure 4). This implies that agents process, in procedure *assign_CPA*, only consistent CPAs. Since the processing of CPAs in this procedure is the only means for extending partial assignments, the following lemma holds:

Lemma 1 Concurrent Search *extends only consistent partial assignments. The partial assignments are received via a CPA, extended and sent forward by the receiving agent.*

The following theorem derives immediately from Lemma 1.

Theorem 1 Concurrent Search *is sound.*

The only lines of the algorithm that report a solution are lines 3, 4 of procedure *assign_CPA*. These lines follow a consistent extension of the partial assignment on a received CPA. It follows that a solution is reported *iff* a CPA includes a complete and consistent assignment. □

To prove completeness for *Concurrent Search*, one needs first to eliminate the stopping condition for the first solution (lines 3-5 of function *assign_CPA* in Figure 4). Another important point is the exact manner in which domains of values of variables are scanned, for the next consistent assignment. Values for assignment are selected only in line 1 of the function *assign_CPA*. For the completeness proof one naturally assumes that the function $assign\_local$, that is run by every agent, scans all values of the current domain exactly once. This is equivalent to the common assumption in all exhaustive backtracking algorithms that all values are tried until a consistent assignment is found (cf. [9]).

With the above assumptions, the completeness of *Concurrent Search* is established in three steps. First, for the case of a single CPA. Then, for several CPAs generated by the IA. Finally, for dynamic generation of CPAs during search. The following lemma establishes the completeness of the *1-CPA* case.

Lemma 2 Concurrent Search *sends forward in a CPA* every *consistent partial assignment.*

To prove Lemma 2, one proceeds in analogy to the proof of completeness for centralized backtrack by Kondrak and van Beek [9]. With no loss of generality assume that every agent holds one variable. Assume that there is some consistent assignment $(X_1, X_2, \dots X_k)$ of length k, that is not received by any agent. Take the highest $j < k$, such that assignment $(X_1, X_2, \dots X_{j-1})$ is sent forward (by agent $j-1$) on a CPA that is received by agent j. There is at least one, sent by the initializing agent. Agent j, has a consistent assignment $(X_1, X_2, \dots X_j)$ that extends $(X_1, X_2, \dots X_{j-1})$, being a sub-tuple of $(X_1, X_2, \dots X_k)$. When agent j extends the received CPA, it succeeds in a consistent partial assignment $(X_1, X_2, \dots X_j)$ and sends it forward. This can be seen clearly in lines 1, 2, 7 of function $assign\_CPA$ in Figure 4. This contradicts the above assumption on the maximality of the assignment $(X_1, X_2, \dots X_{j-1})$, that is sent forward. □

To complete the correctness proof one needs also to show that *Concurrent Search* terminates. The messages of *Concurrent Search* carry CPAs and move either forward or backward. The number of backward moves is finite, since each backward move deletes a value from the domain of the receiving agent (lines 10-11 of function $receive\_CPA$ in Figure 4). To prove termination one needs to show that there can only be a finite number of forward moves (i.e. carrying CPAs). Every agent keeps its current domain in the SP structure and scans its values exactly once, for every different partial assignment received on a CPA. Every move forward carries a consistent partial assignment (by Lemma 2). There is a finite number of different consistent partial assignments, hence a finite number of forward moves in *Concurrent Search*.
Theorem 2 follows immediately.

Theorem 2 *The 1-CPA version of* Concurrent Search *is complete and terminates.*

Having shown the correctness of *Concurrent Search* for a single CPA, one needs to show correctness for the more general case of multiple CPAs generated at the algorithm start.

Theorem 3 *A version of* Concurrent Search *which includes a single split into k search processes at the beginning of the search is complete and terminates.*

Consider a CPA, C_i, that corresponds to a partial domain of one variable of the initializing agent and is passed through the network of all agents. Each agent A_j it passes through generates a data structure SP_i with all domains of its local variables (lines 2, 3 of procedure *receive_CPA*). The only difference between the data structures corresponding to C_i and those that are generated for a 1-CPA version of *Concurrent Search* is in the structure SP_i of the initializing agent. In every other agent, the data structure SP_i and the code it runs are exactly equal to those run for *Concurrent Search* with one CPA. For agents different than the IA, the search procedure of C_i scans exactly the same subspace that is scanned for the one-CPA version of *Concurrent Search*. Consequently, the search procedure corresponding to C_i is correct.

The union of all domain values of the selected variable for a split (in the IA) is exactly equal to the original domain of values of that variable. As shown above, the search sub-trees spanned by all agents that are not the IA, are equal to those spanned for the *1-CPA* algorithm. Each of those equal search subspaces is scanned completely and correctly and all these scans terminate and are performed for every value of the variable of the IA that was selected for the split operation. Consequently, the union of the sub-trees that corresponds to each of the *CPA*s is exactly equal to the search tree that is spanned by the one-*CPA* version of *Concurrent Search*. □

The final step in the correctness proof of *Concurrent Search* is to show that a dynamic split operation does not interfere with the correctness of the algorithm.

Theorem 4 Concurrent Search *with dynamic splitting is complete and terminates.*

Consider agent A_i which is not the initializing agent, that receives a $split\_msg$ and runs the procedure $perform\_split$. It sends forward one or more consistent CPAs that represent non intersecting sub-search-spaces. The completeness and termination of the search on each of these sub-search-spaces follows from the completeness of the search initialized by any CPA of an initializing agent. Agent A_i will declare *no solution* by sending a backtrack message, only after all of its split-*SPs* failed (lines 1, 2 of procedure *backtrack*). In other words, backtracking from multiple CPAs preserves completeness at the splitting agent. The condition to receive failure messages for all values for which a CPA was generated ensures that backtrack corresponds exactly to the case where there is no solution in the scanned search space. The sum of the number of tuples explored in the split search space is equal to the number of tuples in the original search space and therefore the algorithm termination is not affected by the split. □

For the completeness of $ConcDB$ one needs to show also that the additional mechanism for terminating unsolvable search processes on unsolvable sub-search-spaces does not terminate a search-process which explores a sub-search-space that includes a solution. To do so we continue as follows. In every sub-search-space, all tuples of assignments share the head (or prefix) of the assignment. Thus for every sub-search-space we define:

Definition 1 *A* Common Head *(CH) is the maximal prefix of assignments which is included in all partial assignments in a sub-search-space.*

Lemma 3 *A sub-search-space whose CH includes an inconsistent subset of assignments does not include a solution to the DisCSP.*

The proof of lemma 3 derives from the method of constructing an inconsistent assignment in dynamic backtrack [7]. A partial assignment is declared inconsistent only if it causes an empty domain in one of the variables. This implies that this partial assignment cannot be part of a solution. From definition 1 we derive that if a CH includes an inconsistent partial assignment it must be included in all the assignments in its related sub-search-space which means that none of these assignments is a solution to the $DisCSP$. □

Theorem 5 *$ConcDB$ does not terminate search-processes which lead to a solution.*

$ConcDB$ terminates a search process by sending forward unsolvable messages (line 19 in function *receive_CPA*, line 5 in function *mark_unsolvable*, line 3 in function *check_SPs*). Only SPs that have a CH that is an extension of the CH that was found inconsistent are marked unsolvable. The search on these SPs is terminated when the agent receives the CPA corresponding to the unsolvable SP (lines 2,3 of function *assign_CPA*). Lemma 3 implies the proof for Theorem 5. It is immediately clear from Theorem 5 that all partial assignments that lead to a solution will be extended, which implies the completeness of $ConcDB$. □

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, 20], and communication load, in the form of the total number of messages sent [11]. Comparing the number of non-concurrent steps of computation of several search algorithms on $DisCSPs$, measures the concurrency of the algorithms.

Non-concurrent computation effort, in systems with no message delay, are counted by a method similar to that of Lamport's logical clocks [10, 12]. Every agent holds a counter of constraint checks performed. 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 Lamport's logical time [10]. This measure can be upgraded to count the number of non-concurrent constraint checks performed ($NCCC$s), thus incorporate the local computational effort of agents in each step [12].

The experimental evaluation includes four sets of experiments. In the first set the effect of concurrent computation is evaluated by comparing different versions of concurrent search, that have different levels of concurrency. In the second set of experiments the run of $ConcDB$ is compared to the three best performing search algorithms, synchronous conflict backjumping (CBJ) [3, 23], asynchronous forward-checking (AFC) [13] and asynchronous backtracking (ABT) [20, 1]. The third set of experiments checks the behavior of *Concurrent Search* and the other algorithms in real world systems with random message delays. The last set of experiments investigates the heuristic used to determine the level of concurrency of the $ConcDB$ algorithm. The algorithm is run using various *steps_limit* values with and without message delays.

All experiments were conducted using 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 [15, 18]. All four sets of experiments, were conducted on networks with 15 agents ($n = 15$) and 10 values for each agent's variable ($k = 10$). The only exception are the experiments in Section 5.2 which compare the run of the algorithms on problems of different sizes, where problem sizes reach $n = 20$. For each pair of density and tightness values ($p1$, $p2$) 50 different random problems were solved by each algorithm and the results presented are the average of these 50 runs.

5.1 Evaluation of concurrency within $ConcBT$

To investigate the effect of concurrency, one needs to compare the performance of *Concurrent Search* with and without splitting and dynamic splitting. To this end, the simplest concurrent search algorithm, $ConcBT$, was run in a 1-*CPA* version, 5-*CPA* version and a version which performs dynamic re-splitting (using a *step_limit* of 35). The $ConcBT$ algorithm is used in this set instead of $ConcDB$ to eliminate the effect of *Dynamic Backtracking* on the results. The 1-*CPA* version is completely sequential and serves as the baseline for comparison to the concurrent versions.

In the first set of experiments the density of the constraint networks is ($p_1 = 0.7$). The value of tightness, p_2 was varied between 0.1 and 0.9, to cover all range of problem difficulty. Results show averages over 50 runs.

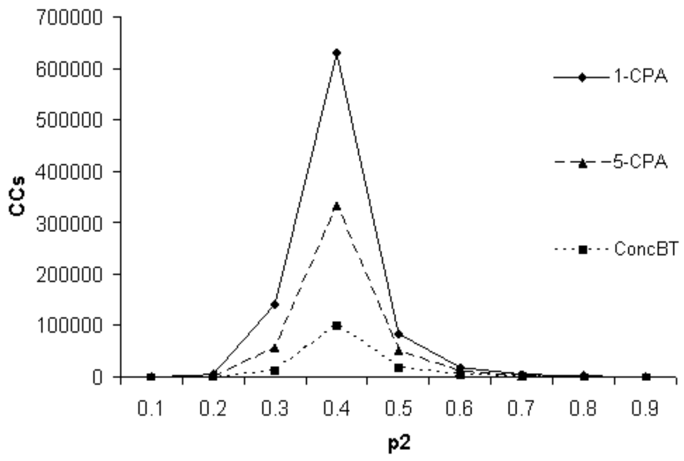

Figure 8: Number of non-concurrent constraint checks in different versions of ConcBT.

Figure 8 shows the computational effort, the number of non-concurrent constraint checks, for all three versions. It is easy to see that concurrency improves the search efficiency and that dynamic re-splitting improves it further. For the harder problem instances the improvement is by a factor of 6 over the 1-CPA version and a factor of 3 over the 5-CPA version. Figure 9 shows the results in total number of messages sent. Clearly, the concurrent versions, either the 5-CPA version or the re-split one, circulate more $CPAs$ in the network. However, the interesting result is that even though $ConcBT$ with dynamic splitting increases the number of traversing CPAs during search, the effect on the total number of messages is negligible. The

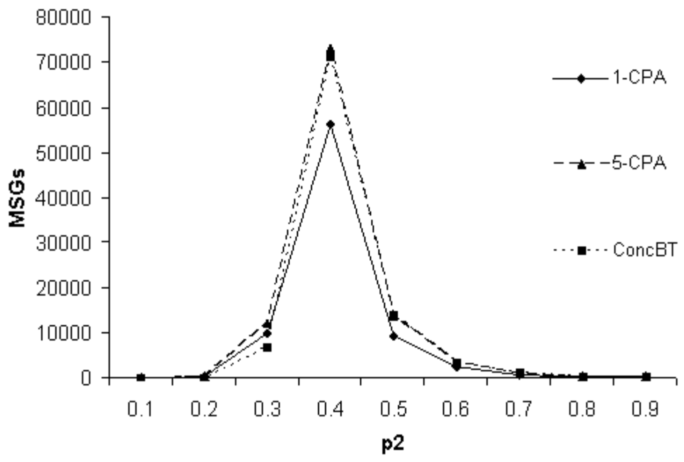

Figure 9: Total number of messages sent in different versions of ConcBT.

dynamic splitting $ConcBT$ does send more messages concurrently but does so during a shorter period of time, resulting in a low amount of total communication.

5.2 Comparing to other DisCSP algorithms

In order to evaluate the performance of concurrent dynamic backtracking (*ConcDB*) it is compared to representatives of the two families of algorithms in the literature. For sequential assignment (synchronous) $DisCSP$ algorithms *Conflict-based Backjumping* is selected [23, 3]. CBJ is an improved version of synchronous backtracking [20], in which agents process *conflict sets* in order to backtrack directly to the culprit agent.

Asynchronous Forward-checking (AFC) [13] is an algorithm which performs sequential assignments like CBJ. A special message which carries the current partial assignment (CPA) is passed among agents and serves as a token to synchronize agent's assignments. An agent which successfully assigned its variable and added its assignment to the CPA, beside sending the CPA forward for the next agent to assign it, sends copies of the CPA to all unassigned agents which perform forward checking concurrently and asynchronously. As a result, the agents detect early a need to backtrack. Agents initiate a backtrack procedure by sending *Not_OK* messages to all agents which may hold the CPA. An agent that receives a *Not_OK* message and then the inconsistent CPA, backtracks as in CBJ by sending the CPA back to the culprit agent.

Asynchronous Backtracking (ABT) [20, 1] is the best performing asynchronous backtracking algorithm. 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 [1]. Based on Yokoo's suggestions [20], agents read in every step, all messages received before performing computation. This forms the best performing version of ABT.

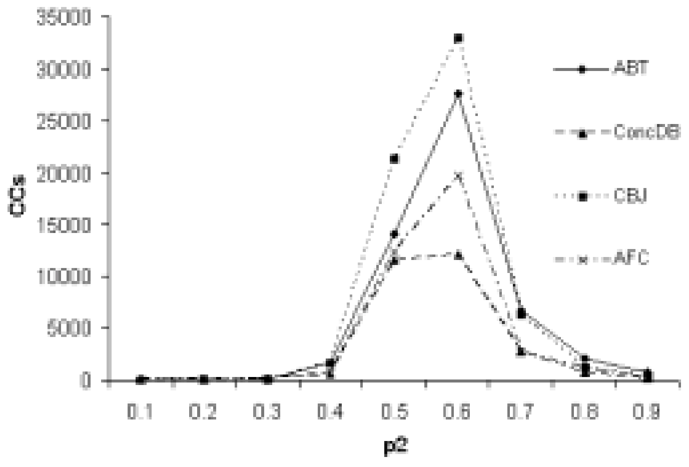

Figure 10: Number of non-concurrent constraint checks performed by ConcDB, ABT and CBJ on low density DisCSPs.

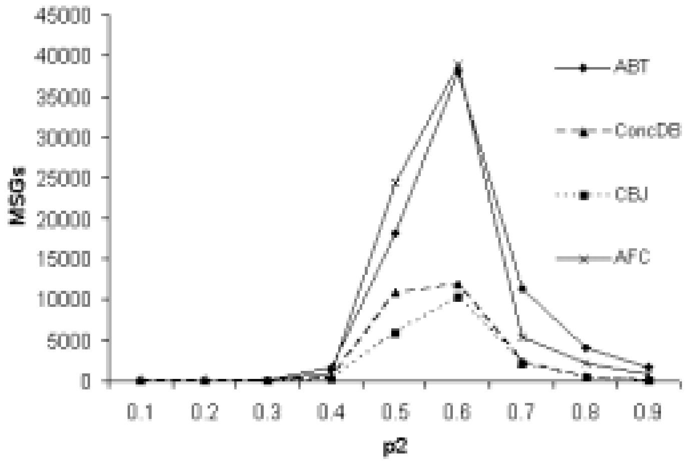

Figure 11: Total number of messages sent by ConcDB, ABT and CBJ on low density DisCSPs

In [20], Yokoo reports that the Asynchronous Weak Commitment algorithm (AWC) is faster than ABT. However, in order to be complete AWC requires agents to hold exponential space (for $Nogoods$). This makes AWC unfeasible for hard instances of large $DisCSPs$. According to Yokoo, this problem can be solved by limiting the size of the $Nogood$ storage, making. a non-complete algorithm.

Figure 10 presents the number of non-concurrent constraint checks performed by $ConcDB$, CBJ, AFC and ABT on problems with low constraint density ($p_1 = 0.4$). For the harder problem instances, *ConcDB* outperforms AFC by a factor of 1.5, ABT by a factor of 2.5 and CBJ by a factor of 3. Figure 11 presents the total number of messages sent by the algorithms in the same run. When it comes to network load the

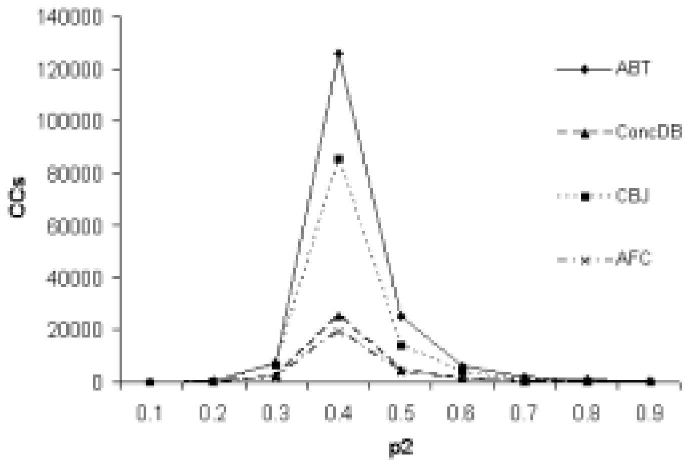

Figure 12: Number of non-concurrent constraint checks performed by ConcDB and ABT on high density DisCSPs.

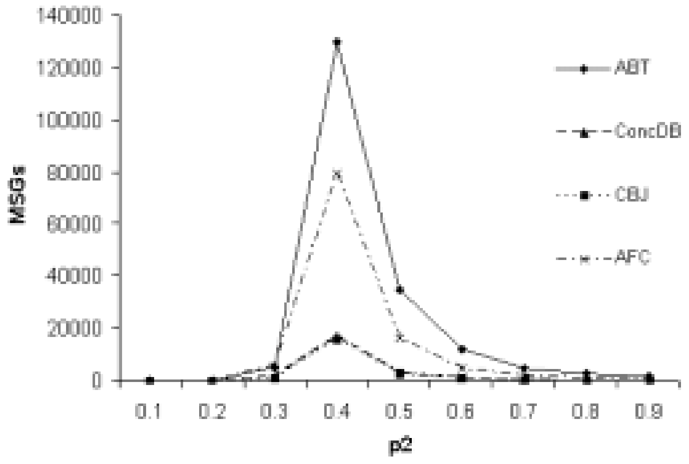

Figure 13: Total number of messages sent by ConcDB, ABT, CBJ and AFC on high density DisCSPs

advantage of $ConcDB$ over ABT and AFC is larger (a factor of 4). As expected, the total network load of the synchronous algorithm, which maintains a single message throughout the search, is the smallest. Still, the total number of messages sent by CBJ and $ConcDB$ are very close.

Figures 12 and 13 show similar results on $DisCSPs$ with higher density ($p_1 = 0.7$). The advantage of $ConcDB$ over ABT and CBJ in $NCCC$s is more pronounced on higher density $DisCSPs$. On the hardest instances, $ConcDB$ performs 6 times less non concurrent constraint checks than ABT and 4 times less than CBJ. AFC also performs better on dense $DisCSPs$. On the hardest instances it performs slightly

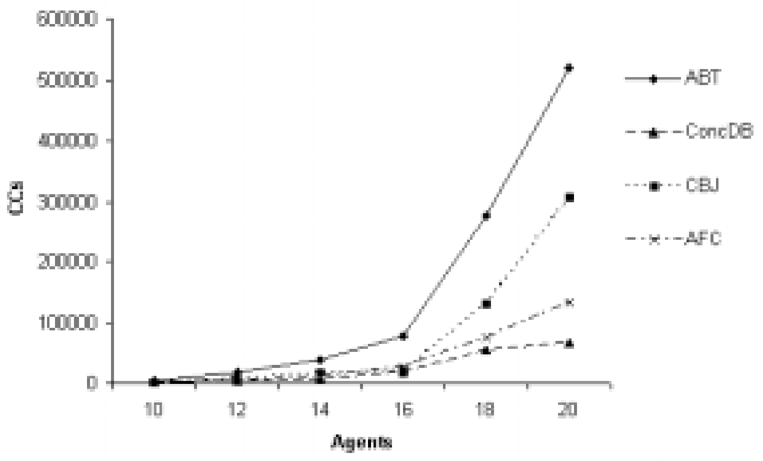

Figure 14: Number of $NCCCs$ performed by ConcDB, ABT CBJ and AFC on the hardest instances of DisCSPs with increasing sizes ($p_1 = 0.4$)

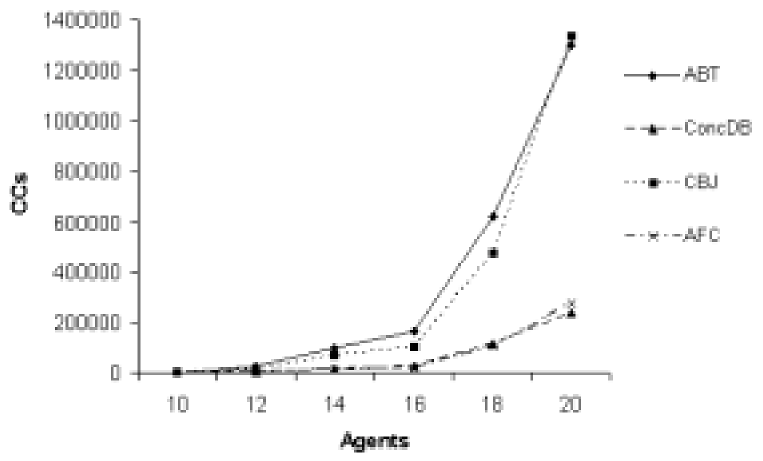

Figure 15: Number of $NCCCs$ performed by ConcDB, ABT CBJ and AFC on the hardest instances of DisCSPs with increasing sizes ($p_1 = 0.7$)

better than $ConcDB$. The total number of messages sent by $ConcDB$ and CBJ are very close. However, AFC sends 4 times more messages and ABT sends 6 times more messages than bothCBJ and $ConcDB$.

Figures 14 and 15 present the number of non concurrent constraint checks performed by the different algorithms on problems with increasing sizes (number of agents). For every size, the result of the most hardest instances $DisCSPs$ are presented. Clearly the performance of ABT and CBJ deteriorates faster than that of AFC and $ConcDB$ when the number of agents increments.

5.3 Performance in the presence of message delays

An important part of the experimental evaluation is to measure the impact of imperfect communication on the performance of distributed search on $DisCSPs$. Message delay has the potential of changing the behavior of distributed search algorithms [5]. For the simplest possible algorithm, synchronous backtracking (SBT) [20], 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. This is true also for versions of synchronous backtracking that perform backjumping [23, 3].

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 a simulator is used, which counts message delays in terms of computation steps and adds them to the accumulated run-time when no computation is performed concurrently [26].

In order to simulate message delays, all messages are delivered by a dedicated $Mailer$ thread. The mailer holds a counter of non-concurrent constraint checks performed by agents in the system. This counter represents the logical time of the system [10] and is called the *Logical Time Counter* (LTC). Every message delivered by the mailer to an agent, carries the LTC value of its delivery to the receiving agent. An agent that receives a message updates its counter to the maximum value between the received LTC and its own value. Next, it performs the computation step, and sends its outgoing messages with the value of its counter, incremented by the number of CCs performed during the step.

The mailer simulates message delays in terms of non-concurrent constraint checks. 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. Then a delay for the message (in number of $NCCCs$) is selected. Each message is assigned a $delivery\_time$ which is the sum of the current value of the LTC and the selected delay (in CCs), and placed in the $outgoing\_queue$. The $Mailer$ delivers 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 delivers the first message.

The non-concurrent run time reported by the algorithm, is the largest LTC value that is 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, constraint checks that were performed concurrently are not counted twice. The actual computational cost during any step is in principle different for different $DisCSP$ algorithms. Measuring non-concurrent constraint checks also enables to evaluate algorithms in which agents perform computation which is not triggered or followed by a message.

Figures 16 and 17 present the results of the third set of experiments in which the four algorithms were run on systems with random message delays. Each message was delayed between 10 to 50 non concurrent constraint checks and the results are presented for low and high density constraint networks. As expected, CBJ is affected most when messages are delayed. In a sequential assignments (synchronous) algorithm there is no concurrent computation by agents. Therefore, each message delay is added to the final run-time result.

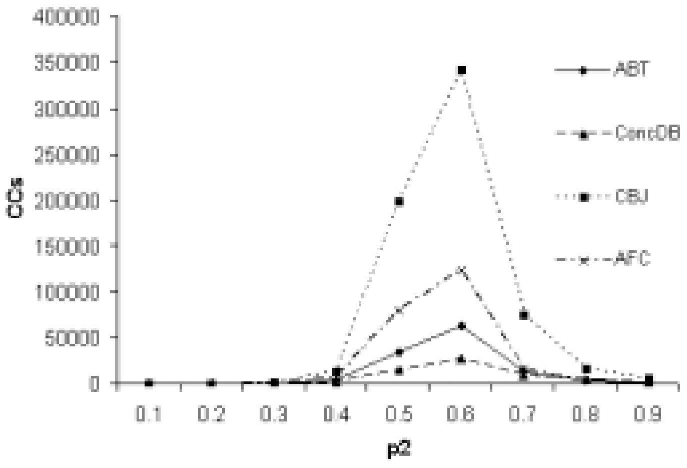

Figure 16: Number of $NCCCs$ performed by ConcDB, ABT, CBJ and AFC on low density *DisCSPs* with random message delay($p_1 = 0.4$)

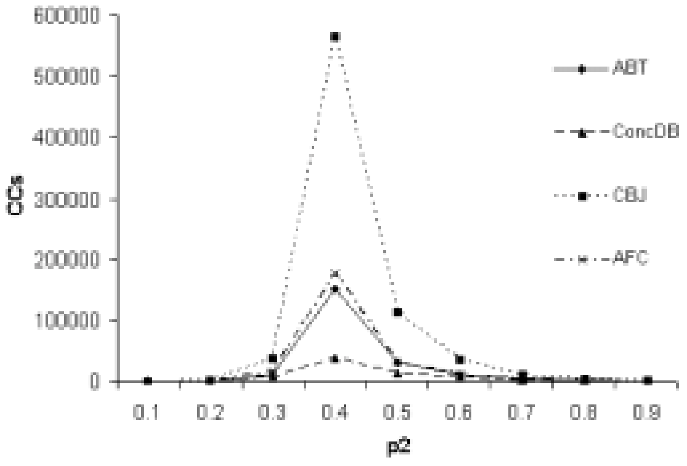

Figure 17: Number of $NCCCs$ performed by ConcDB, ABT, CBJ and AFC on low density *DisCSPs* with random message delay($p_1 = 0.7$)

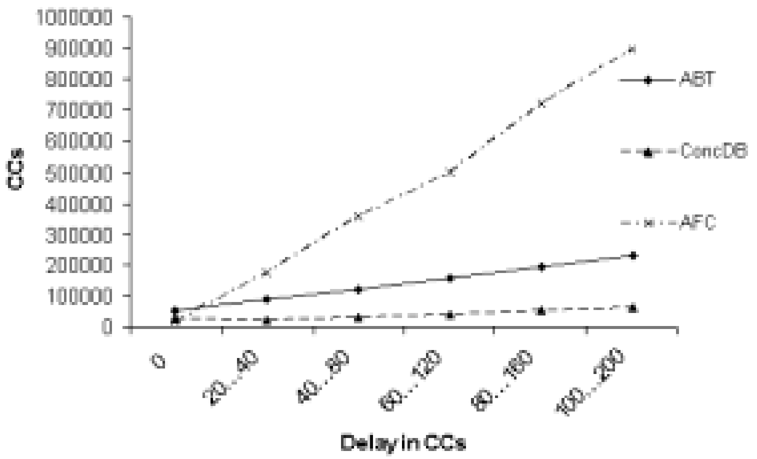

Figure 18: Number of non-concurrent constraint checks performed by ABT, AFC and ConcDB on systems with increasing message delays

Message delays have also a strong effect on AFC since, although it performs concurrent computation, its assignments are performed sequentially. The performance of asynchronous backtracking also deteriorates in the presence of random message delays, while the effect on concurrent search is minor. The advantage of concurrent search over both synchronous and asynchronous backtracking in the presence of message delay is connected to the properties of these algorithms. Previous studies report that ABT performs best when it reads multiple messages before performing computation [23, 1]. When messages are randomly delayed, agents in ABT are more likely to perform computation triggered by a single message. This explains the deterioration in performance of ABT in the presence of random message delays.

To understand the robustness of *Concurrent Search* to message delay imagine the following example. Consider the case where $ConcDB$ splits the search space multiple times and the number of CPAs is larger than the number of agents. In systems with no message delays this would mean that some of the CPAs are waiting in incoming queues, to be processed by the agents. This delays the search on the sub-search-spaces they represent. In systems with message delays, this potential waiting is caused by the system. By choosing the right $steps\_limit$, agents can be kept busy at all times, performing computation against consistent partial assignments.

To further investigate the different behavior of the four algorithms in the presence of imperfect communication, it is interesting to examine the algorithms reaction to message delays of different sizes.

The effect on synchronous CBJ is linear (as expected) and its slope is as steep as the size of the delay. This makes it impossible to display synchronous search together with ABT, AFC and $ConcDB$. Figure 18 presents the impact, in number of non-concurrent constraint checks, of different sizes of random message delays, on Asynchronous Backtracking, Asynchronous Forward-checking and on Concurrent Dynamic Backtracking. As expected, AFC is affected most when the average size of delays increases. The difference in robustness between ABT and $ConcDB$ is striking. While ABT performs a linearly growing

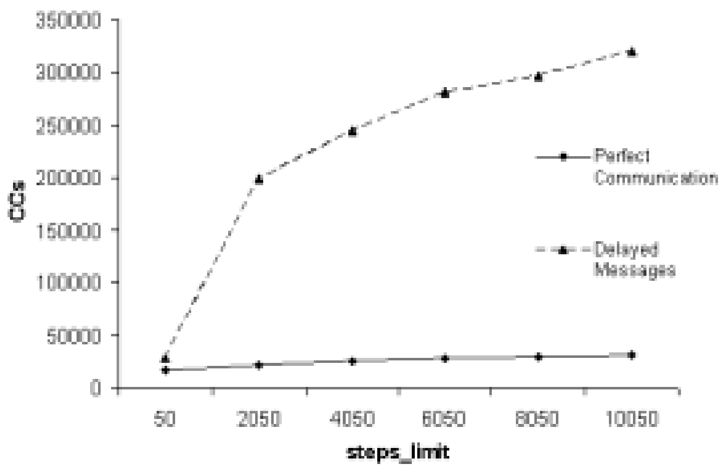

Figure 19: Number of non-concurrent constraint checks performed by ConcDB for increasing $steps\_limits$ ($p_1 = 0.4$)

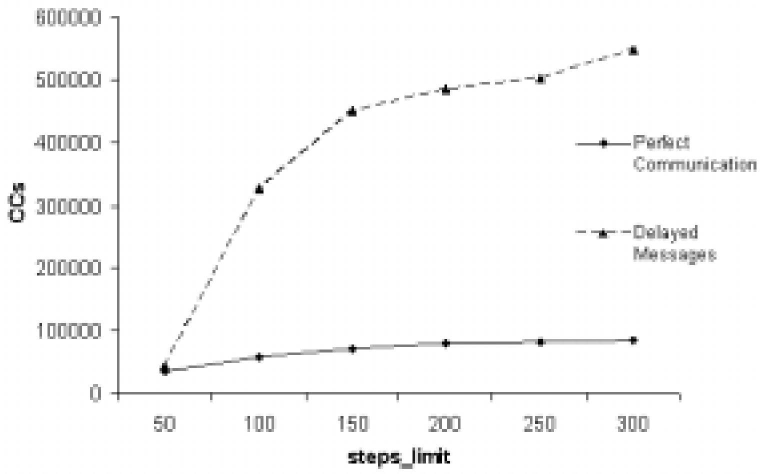

Figure 20: Number of non-concurrent constraint checks performed by ConcDB for increasing $steps\_limits$ ($p_1 = 0.7$)

number of $NCCCs$, $ConcDB$ remains relatively constant. Over a range of average random delay of 200 CCs, $ABT's$ performance deteriorates by a factor of 4.5, while for $ConcDB$ the increase is very slow and the overall factor is about 1.5.

5.4 The impact of the *steps_limit*

The level of concurrency of the $ConcDB$ algorithm is determined by the heuristic upon which the agents decide when to send a split message. Using the heuristic suggested in Section 3, the concurrency of the

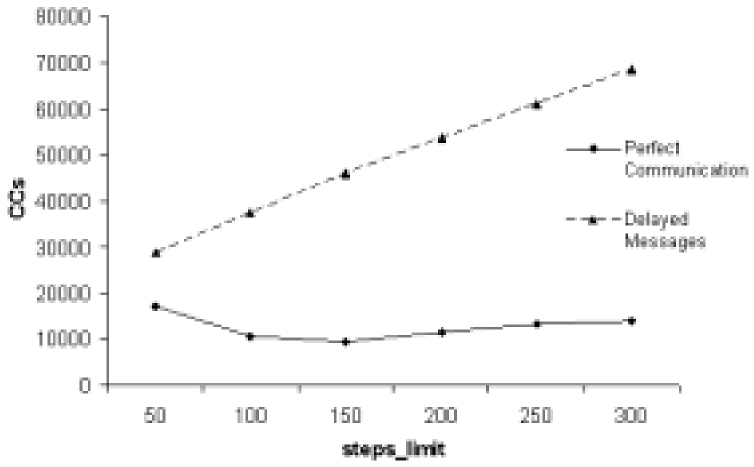

Figure 21: Number of non-concurrent constraint checks performed by ConcDB for a smaller range of $steps_limit$s ($p_1 = 0.4$)

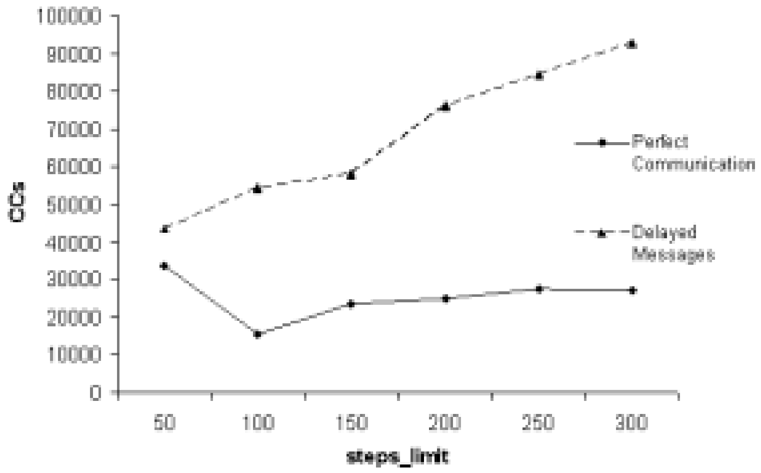

Figure 22: Number of non-concurrent constraint checks performed by ConcDB for a smaller range of $steps_limit$s ($p_1 = 0.7$)

algorithm can be controlled by selecting a good value to the *steps_limit*. Figures 19 and 20 present the number of non concurrent constraint checks performed by $ConcDB$ using various *steps_limit*s. The algorithm was tested with and without message delay. It is clear from the above figures that the choice of a large *steps_limit* deteriorates the performance of $ConcDB$ when run on systems with message delays. The effect on $ConcDB$ running with optimal communication is quite small. These results are not surprising considering the results for CBJ and $ConcDB$ presented in Section 5.2. When the value of the *steps_limit* increases, the level of concurrency decreases and the behavior of the algorithm is closer to the behavior of a single sequential search procedure - CBJ. As for CBJ, it performs well when there are no message delays but performs poorly in the presence of message delay. Figures 21 and 22 present the performance of the algorithm for smaller *steps_limit*s. The stronger effect of the growing *steps_limit* in the presence of message delay is clear in these figures. An interesting observation is that with no message delays, the highest level of concurrency does not produce the best performance.

6. Conclusions

Search algorithms on $DisCSPs$ can be categorized into two families. Single search process algorithms (SPAs) and multiple (concurrent) search process algorithms (MPAs). MPAs are also called *concurrent search* algorithms (CSAs). The state of single process algorithms is defined by a *single tuple of assignments*, one for each agent. When this set of assignments is complete (containing assignments to all variables of all agents) and consistent, the SPA stops and reports a solution. Single search process algorithms can be asynchronous, like ABT [20, 1] or synchronous (SBT [20], CBJ [23, 3]). In concurrent search, multiple concurrent processes search non intersecting parts of the global search space of a $DisCSP$ ([22, 8, 17]). All agents in a MPA participate in every search process, since each agent holds some variables of the search space. As a result, concurrent search is an asynchronous distributed process.

The Concurrent Dynamic Backtracking search algorithm ($ConcDB$) provides an efficient method for several search processes to search concurrently for a solution to a $DisCSP$. The independent random ordering of search on multiple search processes generates an efficient randomization that improves the overall performance. Through a mechanism of dynamic splitting, the number of search processes can be enhanced in some sub-search spaces, thus achieving load balancing in a natural way. The addition of Dynamic Backtracking to concurrent search, enables early termination of search processes on sub-spaces which do not lead to a solution. An inconsistent subset can be found in one sub-space that rules out other sub-spaces as unsolvable. Dynamic backtracking was found to account for $\sim 10\%$ of search processes termination in the experiments of section 5.2

The experimental behavior of $ConcDB$ on random $DisCSPs$ clearly indicates its efficiency, compared to algorithms of a single search process like CBJ, AFC and ABT. This advantage is more pronounced on realistic systems with random message delays where the performance of single process algorithms deteriorates while $ConcDB$ is robust.

References

[1] 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.

[2] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Proc. Workshop on Distributed Constraint of IJCAI01*, 2001.

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

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

[5] 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.

[6] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 179–193, 1996.

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

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

[9] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.

[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, 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.

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

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

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

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

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

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

[19] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96, (short paper)*, pages 561–2, Cambridge, Massachusetts, USA, October 1996.

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

[21] 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.

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

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

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

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

[26] R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *Proc. 5th workshop on distributed constraints reasoning, DCR-04*, Toronto, 2004.