Concurrent Dynamic Backtracking for Distributed CSPs ⋆

Roie Zivan and Amnon Meisels
{zivanr,am}@cs.bgu.ac.il

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

Abstract. A distributed concurrent search algorithm for distributed constraint satisfaction problems (*DisCSP*s) is presented. Concurrent search algorithms are composed of multiple search processes (*SP*s) that operate concurrently and scan non-intersecting parts of the global search space. 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 dynamic backtracking, a search space scanned by one search process can be found unsolvable by a different search process. This enhances the efficiency of the $ConcDB$ algorithm. Concurrent search is an asynchronous distributed algorithm and is shown to be faster than asynchronous backtracking (ABT). The network load of $ConcDB$ is also much lower than that of ABT.

1 Introduction

Distributed constraint satisfaction problems (*DisCSP*s) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [Yokoo2000,Solotorevsky et. al.1996]). 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 [Meseguer and Jimenez2000,Bessiere et. al.2001].

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 [Solotorevsky et. al.1996].

⋆ Partially supported by the Lynn and William Frankel Center for Computer Science

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

One method for achieving concurrency in search on Distributed CSPs is to enable agents to cooperate in a single backtrack procedure. In order to avoid the waiting time of a single backtrack search, agents compute assignments to their variables asynchronously. In asynchronous backtracking algorithms, agents assign their variables without waiting to receive information about all relevant assignments of other agents [Yokoo et. al.1998,Silaghi2002]. In order to make asynchronous backtracking correct and complete, all agents share a static order of variables and the algorithm keeps data structures for nogoods that are discovered during search (cf. [Bessiere et. al.2001]). The present paper proposes a different way of achieving concurrency for search. In order to achieve shorter overall runtime, 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 [Zivan and Meisels2002]. Agents and variables are ordered randomly on each of the search processes, diversifying the sampling of the search space. Agents generate and terminate search processes dynamically during the run of the algorithm, thus creating a distributed asynchronous algorithm [Zivan and Meisels2004]. The degree of concurrency during search changes dynamically and enables automatic load balancing (see section 2.1).

The present paper proposes Concurrent Dynamic Backtracking $ConcDB$ that performs dynamic backtracking on each of its concurrent sub-search spaces [Ginsberg1993]. Since search processes are dynamically generated by $ConcDB$, the performance of backjumping in one search space can indicate that other search spaces are unsolvable. This feature, combined with the random ordering of agents in each search process, enables early termination of search processes discovered by DB to be unsolvable.

The principles and mechanism of *Concurrent Search* along with a detailed description of the $ConcDB$ algorithm are described in section 2. A correctness and completeness proof for $ConcDB$, is outlined in section 3. Section 4 presents an extensive experimental evaluation, which demonstrates multiple advantages of $ConcDB$. A comparison of $ConcDB$ to asynchronous backtracking (ABT) [Yokoo2000,Bessiere et. al.2001] is presented in section 4.1. For all measures of concurrent performance, from number of steps through number of concurrent constraints checks, to number of messages sent, *ConcDB* outperforms *ABT* and its advantage is more pronounced for harder problem instances. For all three measures of performance, the difference between $ConcDB$ and ABT *grows with message delay*. In other words, concurrent dynamic backtracking is more robust to message delay than asynchronous backtracking [Fernandez et. al.2002].

2 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 space 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 (SP)*s, 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 *CPA* (Current Partial Assignment) data structure. Each *CPA* represents one search process, and holds the agents' current assignments in the corresponding search process. An agent that receives a *CPA* tries to assign its local 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*.

An agent can generate a set of CPAs that split the search space of a single CPA that passed through that agent, by splitting the domain of one of its variables. Agents can perform splits independently and keep the resulting data structures (*SP*s) privately. All other agents need not be aware of the split, they process all $CPAs$ in exactly the same manner (see section 2.1). *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 the *CPA* in a sub-search-space (without finding a solution), is used to determine the need for re-splitting of search-spaces. This generates a nice mechanism of load balancing, creating more search processes on heavily backtracked search spaces.

A backtrack operation is performed by an agent which fails to find a consistent assignment 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 by performing a backtrack operation.

The search ends unsuccessfully, when all *CPA*s return for backtrack to the IA and the domain of their first variable is empty. 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. The next agent a CPA is sent to is selected randomly. Dynamic splitting of search spaces is performed asynchronously. Consequently, the steps of agents in different search process are interleaved in a non predefined order. This makes Concurrent Search algorithms asynchronous [Lynch1997].

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 (for a detailed example of dynamic splitting see section 2.2).

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. The *ID* of CPAs enables all agents to create *CPA*s independently, with a unique *ID*. This is the basis for dynamic splitting of the search space. When a split is performed during search, all CPAs generated by the agent that performs the split have a unique ID and carry the ID of the CPA from which they were split.

2.1 Description of Concurrent Search

The main functions of concurrent search are presented in Figure 1.

- The main function **Concurrent Search**, initializes the search if it is run by the *initializing agent (IA)*. It initializes the search by creating multiple *SP*s, 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). 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 10-11). If not, the CPA is saved as the original CPA for use in future backtracks.
- **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).
- 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). 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). Then a backtrack message is sent to the agent which its 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

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

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

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

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

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

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

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

Fig. 1. Concurrent Search

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 2 presents an example of a DisCSP, searched concurrently by two processes represented by two CPAs, CPA_1 and CPA_2. Each of the four agents A_1 to A_4, holds two SPs. The current domains of the SPs are shown in Figure 2. The domains on the left represent the state after 3 assignments to CPA_1. The domains on the right of

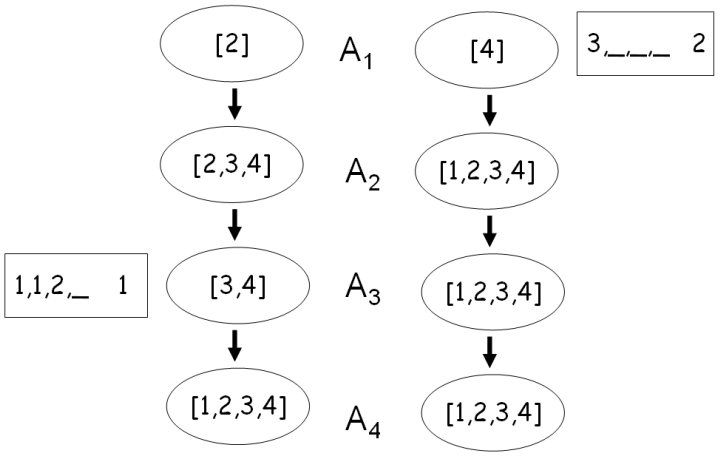


Fig. 2. *Concurrent Search* with two CPAs

figure 2 represent the state after the first assignment to CPA_2. CPA_1 is depicted on the LHS of figure 2 and CPA_2 is on the top RHS. Each CPA has its ID on its right.

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.

2.2 Example of dynamic splitting

Consider the constraint network that is described in figure 3. All three agents own one variable each, and the initial domains of all variables contain four values $\{1..4\}$. The constraints connecting the three agents are: $X_1 < X_2$, $X_1 > X_3$, and $X_2 < X_3$. The initial state of the network is described on the LHS of Figure 3. In order to keep the example small, no initial split is performed, only dynamic splitting. The 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 3. The run of the algorithm during these 5 steps is described in detail below:

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

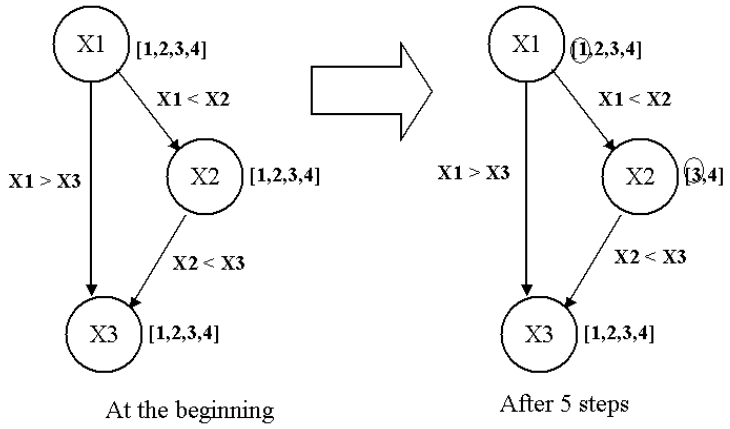


Fig. 3. Initial state and the state after the CPA travels 5 steps without returning to its generating agent

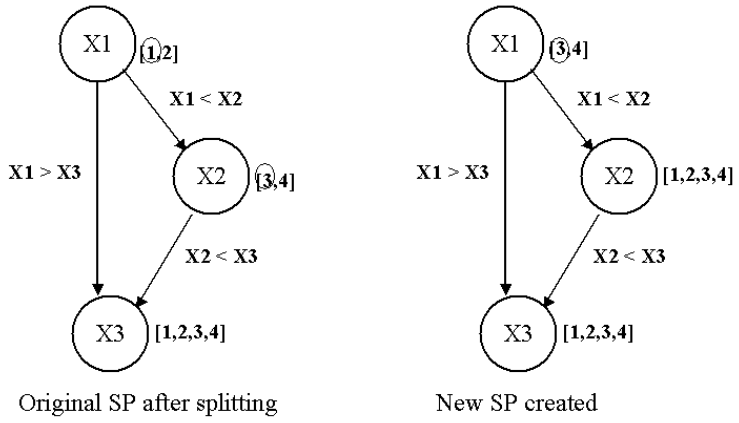


Fig. 4. The new non intersecting search spaces now searched using two different *CPAs*

In the current step of the algorithm, agent X_3 receives a CPA which has reached the $step\_limit$. According to lines 8-9 of function $receive\_CPA$ 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 new domain.

ConcDB:
..
..
..
9. $unsolvable$: mark_unsolvable

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

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

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

check_SPs(inconsistent_assignment)
1. **for each** of the $\{SPs \setminus current\_SP\}$
2. **if**(SP.contains(inconsistent_assignment))
3. send(unsolvable, SP.next_agent)
4. $CPA \leftarrow \{last\_sent\_CPA\} \setminus last\_assignment$
5. rename_SP
6. $assign\_CPA$

Fig. 5. Methods for Dynamic Backtracking

- Creates a new CPA and assigns it with 3 (a value from the new domain).
- Sends the new CPA to a randomly chosen agent.

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

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

2.3 Concurrent Dynamic Backtracking

The method of backjumping that is used in the proposed $ConcDB$ algorithm is based on $Dynamic\ Backtracking$ [Ginsberg1993,Bessiere et. al.2001]. 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 [Ginsberg1993]. 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* [Ginsberg1993,Bessiere et. al.2001]. 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 belong to multiple 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 CPA either belongs or was split from.
- check if the SP was split.
- if it was:
  - send an $unsolvable$ message to the *next_agent* of the related 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, send corresponding $unsolvable$ messages and resume the search on them with new generated $CPAs$.

The change of ID makes the process independent of whether the backtrack message included the *original CPA* or one of its split offsprings.

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.

Agents that receive a CPA first check if the related SP was not marked $unsolvable$. If so they terminate the CPA and its related SP.

Figure 5 presents the methods $ConcDB$, $receive\_CPA$ and **backtrack**, that were changed from the general description of *Concurrent Search* in Figure 1, and two additional methods needed for adding *Dynamic Backtracking* to concurrent search.

In method $receive\_CPA$ a check is made in lines 3,4 if the SP related to the received CPA is marked unsolvable. In such a case the CPA is not assigned and the related SP is terminated. For a backtracking CPA (lines 13-18) a check is made whether the SP was split by agents who received the CPA after this agent (line 16). 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 its related SP (line 18). Before calling $assign\_CPA$, a check is made whether there are other SPs which can be declared unsolvable. This can happen when the head of their partial assignment (their

common header 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 new generated CPA.

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 as unsolvable, and sends unsolvable messages to the agents to whom the CPA of this SP, and any other CPA split from it, were sent. In method $backtrack$, the agent inserts the culprit inconsistent partial assignment into the backtrack message (line 9) before sending it back in line 10.

3 Correctness of Concurrent Search

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 1). This implies that agents process, in procedures *receive_CPA* and *assign_CPA*, only consistent CPAs. Since the processing of CPAs in these procedures are the only means for extending partial assignments, the following lemma holds:

Lemma 1 *$ConcurrentSearch$ algorithms extend only consistent partial assignments. The partial assignments are received via a CPA and extended and sent forward by the receiving agent.*

The correctness of $ConcurrentSearch$ includes soundness and completeness. The soundness of $ConcurrentSearch$ follows immediately from Lemma 1. 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.

In order to prove the completeness of the $ConcDB$ algorithm we first outline the proof for the simpler concurrent backtrack version and then show that adding conflict based backjumping does not affect the completeness of the algorithm. The main points of the completeness proof for general concurrent search are the following:

- Completeness for the case of a single CPA, is equivalent to the proof of completeness for centralized backtrack by Kondrak and vanBeek [Kondrak and vanBeek1997].
- For several CPAs generated by the IA, the only difference from the $1-CPA$ case is in the data structures of the IA.
- A dynamic split operation does not interfere with the completeness of the algorithm.

The above three points were established for $ConcBT$ in [Zivan and Meisels2004]. For the completeness of $ConcDB$ one continues as follows. In every sub search space all tuples of assignments share the head of the assignment. Thus for every sub-search-space we define:

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

Lemma 2 *A sub-search-space whose CH includes an inconsistent subset of assignments does not include a solution to the DisCSP.*

The proof of lemma 2 derives from the method of constructing an inconsistent assignment in dynamic backtrack [Ginsberg1993,Bessiere et. al.2001].

Lemma 3 *$ConcDB$ does not terminate search-processes which lead to a solution.*

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 later terminated. Lemma 2 implies the proof for lemma 3. It is immediately clear from lemma 3 that all partial assignments that lead to a solution will be extended, which implies the completeness of $ConcDB$.

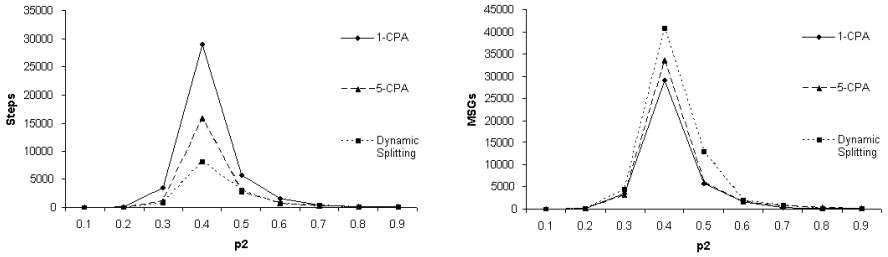
4 Experimental Evaluation

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 [Lynch1997,Yokoo2000], and communication load, in the form of the total number of messages sent [Lynch1997]. Comparing the number of concurrent steps of computation of search algorithms on DisCSPs, measures the time of run of the algorithms.

Concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [Lamport1978,Meisels et. al.2002]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent's counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search, we achieve a measure of concurrent search effort that is close to Lamports logical time [Lamport1978]. If instead of steps of computation we count the number of concurrent constraints check peformed (CCCs), we take into account the local computational effort of agents in each step [Meisels et. al.2002].

An important part of the experimental evaluation is to measure the impact of imperfect communication on the performance of concurrent search. Message delay can change the behavior of distributed search algorithms [Fernandez et. al.2002]. In the presence of concurrent computation, the time of message delays must be added to the total algorithm time *only if no computation was performed concurrently*. To achieve this goal, we use a simulator which counts message delays in terms of computation steps and adds them to the accumulated run-time when no computation is performed concurrently [Zivan and Meisels2004a].

Experiments were conducted on random networks of constraints of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of CSP algorithms cf. [Prosser1996,Smith1996]). All three sets of experiments were conducted on networks with 15 agents ($n = 15$), 10 values for each variable ($k = 10$) and two values of constraints density $p_1 = 0.4$ and



(a) (b)

Fig. 6. (a) Number of concurrent steps for either 1-CPA, 5-CPAs, or dynamic number of CPAs, (b) Total number of messages, sent by the same three versions.

$p_1 = 0.7$ The tightness value p_2, is varied between 0.1 and 0.9, to cover all ranges of problem difficulty.

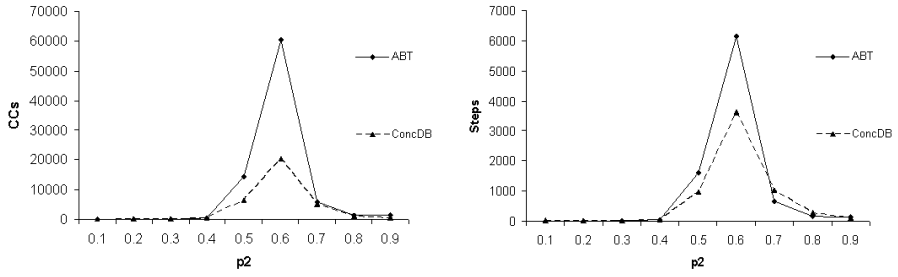
To investigate the effect of concurrency, one needs to compare the performance of *Concurrent Search* with and without splitting and dynamic splitting. To this end, the simplest concurrent search algorithm, $ConcBT$, was run in a 1-*CPA* version, 5-*CPA* version and a 5-*CPA* version with dynamic re-splitting, using a step limit of 35. The 1-*CPA* version is completely sequential and serves as the baseline for comparison to the concurrent versions.

The LHS of figure 6 shows the computational effort in number of concurrent steps to a solution, for all three versions. It is easy to see that concurrency improves the search efficiency and that dynamic resplitting improves it further. The results in concurrent constraints checks are similar and not presented due to limited space. The total number of messages sent by all three versions of the algorithm are presented on the RHS of figure 6. Surprisingly, the effect of dynamic splitting on message load is minor.

4.1 Comparing to Asynchronous Backtracking

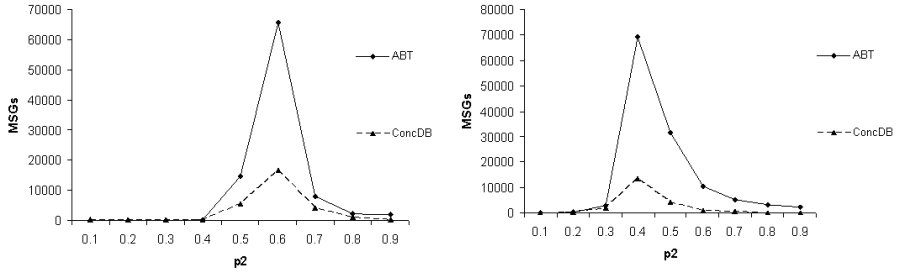
The performance of concurrent dynamic backtracking (*ConcDB*) can be compared to asynchronous backtracking (ABT) [Yokoo2000]. In ABT agents assign their variables asynchronously, and send their assignments in $ok?$ messages to other agents to check against constraints. A fixed priority order among agents 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 our implementation of ABT, the $Nogoods$ are resolved and stored according to the method presented in [Bessiere et. al.2001]. Based on Yokoo's suggestions [Yokoo2000] the agents read, in every step, all messages received before performing computation.

The LHS of figure 7 presents the comparison of the number of concurrent constraints checks performed by $ConcDB$ and ABT on problems with low dinsity ($p_1 = 0.4$). For the harder problem instances, *ConcDB* outperforms ABT by a factor of 3. On the RHS of figure 7 The results are presented in the number of concurrent steps of computation. The smaller factor of difference can be related to the larger amount of



(a) (b)

Fig. 7. (a) Number of concurrent constraints checks performed by ConcDB and ABT, (b) Number of concurrent steps for both algorithms.



(a) (b)

Fig. 8. Total number of messages sent by ConcDB and ABT on DisCSPs with low density (a) and high density (b).

local computation ABT perfoms in each step since it reads all the messages which it received up to this step.

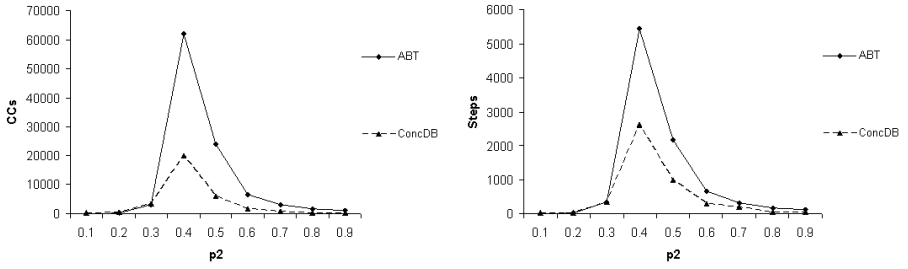
Figure 8 presents the total number of messages sent by both algorithms on DisCSPs with density $p_1 = 0.4$ (LHS) and $p1 = 0.7$ (RHS). In both cases when it comes to network load, the advantage of $ConcDB$ is larger (a factor of 4, for $p_1 = 0.4$ and 5, for $p1 = 0.7$).

Figure 9 presents a comparison of $ConcDB$ and ABT on DisCSPs with higher density ($p_1 = 0.7$). The results are very similar.

Figure 10 presents the results of the set of experiments in which the algorithms were run on a system with random message delay. Each message was delayed between 5 to 10 steps and the results in logical steps are presented for low and high density (LHS and RHS of figure 10 respectively). Random message delay deteriorates the performance of asynchronous backtracking while the effect on concurrent dynamic backtracking is minor. The results in figure 10 show a larger factor of difference between the two algorithms.

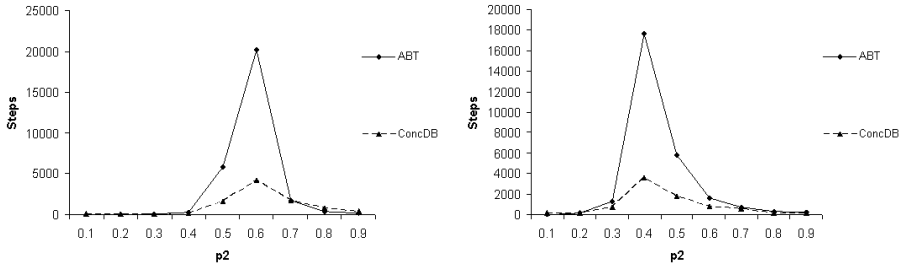
5 Conclusions

Concurrent search on distributed CSPs has been presented in detail. Concurrent search algorithms maintain multiple search processes on non intersecting parts of the global



(a) (b)

Fig. 9. (a) Number of concurrent constraints checks for $p_1 = 0.7$, (b) Number of concurrent steps for $p_1 = 0.7$.



(a) (b)

Fig. 10. (a) Number of logical concurrent steps performed by ConcDB and ABT on low density DisCSPs, (b) on high density DisCSPs.

search space of a DisCSP ([Zivan and Meisels2002,Hamadi2002]). All agents in concurrent search participate in every search process, since each agent holds some variables of the search space. Each agent holds the current domains of its variables, for each of the search processes. Search processes are dynamically generated by agents in an asynchronous distributed process.

Concurrent dynamic backtracking ($ConcDB$) provides an efficient method for several search processes to search concurrently a DisCSP. Dynamic backtracking enables concurrent search an 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 and rule out other sub-spaces as unsolvable. In such a case, the search on the obsolete sub-search-spaces is terminated by an elegant procedure which does not affect viable search processes in progress.

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

Concurrent search, as proposed in the present paper and in [Zivan and Meisels2004], may seem similar to former approaches of parallelism. There is, however, a major difference between Concurrent Dynamic Backtracking $ConcDB$ and the *interleaved parallel search algorithm - IDIBT* [Hamadi2002]. IDIBT runs multiple processes of asynchronous backtracking and its multiplicity is fixed at the start of its run [Hamadi2002]. Dynamic splitting of the search space improves the search by a meaningful factor (see Figure 6). This is in contrast to $IDIBT$, where performance deteriorates for more than 2 contexts [Hamadi2002].

References

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

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

[Ginsberg1993] M. L. Ginsberg Dynamic Backtracking. *Artificial Intelligence Research*,1: 25-46, 1993

[Hamadi2002] Y. Hamadi Interleaved Backtracking in Distributed Constraint Networks. *Intern. Jou. AI Tools*, 11: 167-188, 2002.

[Kondrak and vanBeek1997] G. Kondrak and P. vanBeek A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89: 365-87, 1997.

[Lamport1978] L. Lamport Time, clocks and the ordering of events in a distributed system. *Comm. of ACM*, 21: 558-565, 1978.

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

[Meisels et. al.2002] A. Meisels et. al. Comparing performance of Distributed Constraints Processing Algorithms. *Proc. AAMAS-2002 Workshop on Distributed Constraint Satisfaction*, Bologna, July, 2002.

[Meseguer and Jimenez2000] P. Meseguer M. A. Jimenez Distributed Forward Checking. *Proc. CP-2000 Workshop on Distributed Constraint Satisfaction*, Singapore, 22 September, 2000.

[Prosser1996] P. Prosser An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81-109, 1996.

[Smith1996] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. In *Artificial Intelligence*, 81:155-181, 1996.

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

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

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

[Yokoo2000] M. Yokoo. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents & Multi-Agent Sys.*, 3(2): 198-212, 2000.

[Zivan and Meisels2002] A. Meisels et. al. Parallel Backtrack search on DisCSPs. *Proc. AAMAS-2002 Workshop on Distributed Constraint Satisfaction*, Bologna, July, 2002.

[Zivan and Meisels2004] R. Zivan and A. Meisels. Concurrent Backtrack search on DisCSPs. to apear in *FLAIRS 2004*, May 2004. (full version http://www.cs.bgu.ac.il/˜zivanr)

[Zivan and Meisels2004a] R. Zivan and A. Meisels. Message delay and DisCSP search algorithms. submit to *ECAI 2004*, August 2004.