Concurrent Backtrack Search on DisCSPs

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 search algorithm for solving distributed constraint satisfaction problems (*DisCSP*s) is presented. The proposed algorithm is composed of multiple search processes (*SP*s) that operate concurrently. Concurrent search processes scan non-intersecting parts of the search space. Each *SP* is represented by a unique data structure, containing a current partial assignment (CPA), that is circulated among the different agents. The splitting of the search space, leading to several concurrent *SP*s is achieved by splitting the domain of one or more variables. The proposed algorithm generates concurrent search processes *dynamically*, starting with the initializing agent, but occurring also at any number of agents during search.

The Concurrent BackTracking (ConcBT) algorithm is presented and an outline of the correctness proof is given. Experimental evaluation of the algorithm, on randomly generated *DisCSP*s, is presented. ConcBT outperforms asynchronous backtracking (ABT) (Yokoo2000) on random DisCSPs with different patterns of message delays. Load balancing for ConcBT is achieved by adding concurrent search trees dynamically, performing re-splitting of the search space by the use of a simple heuristic.

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. Following common practice, it is assumed that an agent can send messages to any one of the other agents (Yokoo2000; Meseguer and Jimenez2000; Bessiere et. al. 2001).

Several asynchronous search algorithms for *DisCSP*s have been proposed in the last decade (Yokoo2000; Bessiere et. al. 2001). The common feature of all asynchronous search algorithms is that agents process their assignments asynchronously, even if their local assignments are not consistent with other agents' assignments. In order to make asynchronous backtracking correct and complete the algorithm usually keeps data structures for nogoods (cf. (Bessiere et. al. 2001)).

The present paper proposes a search algorithm on *DisCSP*s that is composed of several concurrent search processes, each of which is a synchronous backtrack procedure. The proposed algorithm uses a form of message in which agents send and receive *a consistent partial assignment of multiple agents*. When such a message includes a complete assignment, to all variables of all agents, the search stops and the solution is reported. Agents that initialize a search process generate a data structure that we term *current partial assignment - CPA*. The initializing agent records on the *CPA* its consistent assignment and sends it to another agent. Each receiving agent adds its consistent assignment if it exists to the *CPA*. Otherwise, it backtracks by sending the same *CPA* to a former agent.

The concurrency of the concurrent backtrack (ConcBT) algorithm is achieved by the circulation of multiple *CPA*s. 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. This changes dynamically the degree of concurrency during search and enables automatic load balancing. The splitting and re-splitting of the search space is performed independently by agents and is thus a distributed process.

The concurrent backtrack (ConcBT) algorithm is described in the following section. A correctness and completeness proof for ConcBT is outlined next. Then we present experimental evaluations, in which the ConcBT algorithm, out performs asynchronous backtracking (ABT), on randomly generated *DisCSP*s in systems with different patterns of message delays. The last section summarize our conclusions.

Concurrent Backtrack - ConcBT

The ConcBT algorithm performs concurrent backtrack searches on disjoint parts of the *DisCSP* search-space. Each agent holds the data relevant to its state on each sub-search-space in a separate data structure which we term *Search Process (SP)*. Agents in the ConcBT algorithm 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 on the *CPA*, using only the current domains in the *SP* related to the received *CPA*. The uniqueness of the *CPA* for every search space ensures that assignments are not done concurrently in a single sub-search-space.

The main point of interest of the ConcBT algorithm, is its ability to split the search space dynamically. Each agent can generate a set of CPAs that split the search space of a 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 **Algorithm description** below). *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. The present study uses a heuristic of counting the number of times agents pass the *CPA* in a sub-search-space (without finding a solution), to determine the need for re-splitting of that sub-search-space. 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 in the search-space corresponding to the partial assignment on the *CPA*. Agents that have performed dynamic splitting, have to collect all of the returning CPAs, of the relevant SP, before performing a backtrack operation.

Main objects of ConcBT

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 (see subsection **Example of dynamic splitting** for a detailed example).

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 *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.

Algorithm description

The ConcBT algorithm is run on each of the agents in the DisCSP and uses the following terminology:

- $CPA\_generator$: Every *CPA* carries the *ID* of the agent that created it.
- *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 return to its generator.
- *split_set*: the set of *SP-ID*s, stored in each SP, including the IDs of the active SPs that were split from the SP by the agent holding 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*.

The messages exchanged by agents in ConcBT are the following:

- **CPA** - a regular CPA message.
- **backtrack_msg** - a CPA sent in a backtrack operation.
- **stop** - a message indicating the end of the search.
- **split** - a message that is sent in order to trigger a split operation. Contains the *ID* of the *SP* to be split.

The ConcBT algorithm is presented in two parts. The first part includes the main function of the algorithm and functions that perform assignments on the CPA when it moves forward (Figure 1).

- The main function **ConcBT**, initializes the search if it is run by the *initializing agent (IA)*. It initializes the algorithm 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.

- **ConcBT**:
  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. $assign\_CPA$
- **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*

Figure 1: Main and Assign parts of ConcBT

- **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).
- **assign_CPA** tries to find an assignment for the local variables of the agent, which is consistent with the assignments on the *current_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 rest of the functions of the ConcBT algorithm are presented in Figure 2.

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

Figure 2: Backtrack and Split for ConcBT

- 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 that were split from the $current\_CPA$ have also failed (line 2). When all split CPAs have returned unsuccessfully, a backtrack message is sent carrying the ID of the $origin\_CPA$. In case of an *IA*, the *origin_SP* is marked as a failure (lines 3-4). If all other *SP*s are marked as failures, the search is ended unsuccessfully (line 6).
- 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 *SP*s $origin\_SP$ (line 6).

The algorithm ends unsuccessfully, when all *CPA*s return for backtrack to the IA and the domain of their first variable is empty. The algorithm ends successfully if *one CPA* contains a complete assignment, a value for every variable in the DisCSP.

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 it's variable with 1, and sends to X_2 a CPA with a step counter $CPA\_steps = 1$.
2. X_2 assigns it's variable with 2, and sends the CPA with both assignments, and $CPA\_steps = 2$, 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 it's variable, with $CPA\_steps = 3$.
4. X_2 reassigns it's variable with the value 3, and sends the CPA again to X_3 after raising the step counter to 4.
5. X_3 receives the CPA with X_2's new assignment.

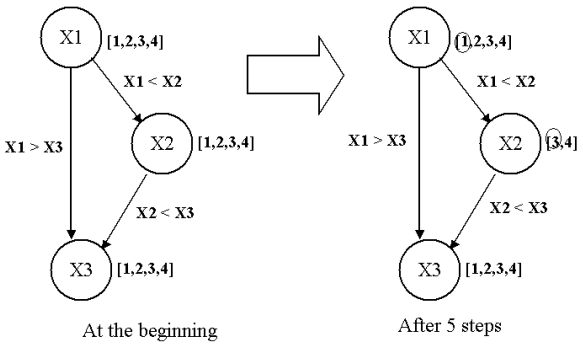

Figure 3: Initial state and the state after the CPA travels 5 steps without returning to its initializing agent

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

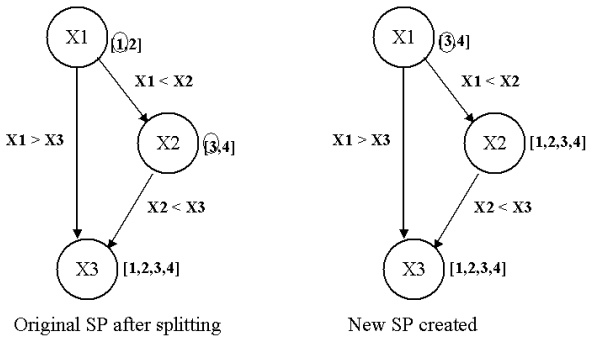

Figure 4: The new non intersecting search spaces now searched using two different $CPAs$

- 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 since they were not consistent with X_1's assignment (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 the CPA check the new assignment against all there (full) domain values (RHS of figure 4).

Correctness of ConcBT

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*. 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 *ConcBT extends only consistent partial assignments. The partial assignments are received via a CPA and extended and sent forward by the receiving agent.*

The correctness of ConcBT includes soundness and completeness. The soundness of ConcBT 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.

The completeness proof for $ConcBT$ will only be outlined here. The main points of the proof are the follow-

ing:

- 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, where the only difference from the $1-CPA$ case is in the data structures of the IA.
- Finally, it is shown that a dynamic split operation does not interfere with the correctness of the algorithm.

The reader is encouraged to look up the full version of the paper (Zivan and Meisels2003) for the full completeness proof.

Experimental Evaluation

The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables. Such uniform random constraints networks of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 are commonly used in experimental evaluations of CSP algorithms (cf. (Prosser1996; Smith1996)). Experiments were conducted on networks with 10 variables ($n = 10$) and 10 values ($k = 10$). In all of our experiments $p1 = 0.7$ and the value of the tightness p_2 is varied between 0.1 and 0.9, to cover all ranges of problem difficulty.

The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance: computation effort, in the form of steps of computation (Yokoo2000), and communication load, in the form of the total number of messages sent (Lynch1997).

Evaluation of concurrency

To investigate the effect of concurrent computation in the ConcBT algorithm one needs to run the algorithm with different number of CPAs, and compare the search effort for finding a solution. The larger the number of CPAs, the greater amount of concurrent exploration of the search space is expected. On the other hand, since the number of search processes operating at a certain time is bounded by the number of agents, a large number of CPAs might increase the time a CPA is waiting at one of the agents message queues.

The ConcBT algorithm was run in a 1-CPA version, 5-CPA version and a 5-CPA version with dynamic resplitting, using a step limit of 20. The 1-CPA version is completely sequential and serves as the base line for comparing the concurrent versions.

Figure 5 shows the computational effort in number of steps for all three cases. It is easy to see that the difference between versions with a constant number of CPAs (1-CPA and 5-CPA) is small. Substantial improvement

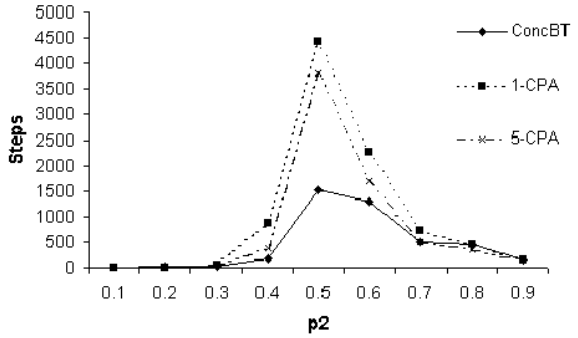

Figure 5: Number of steps in different versions of ConcBT, either 1-CPA, 5-CPAs, or dynamic number of CPAs

is achieved by the use of dynamic re-splitting. For the hardest problem instances (p_2=0.5) dynamic resplitting lowers the computational cost by a factor of 2.5.

Comparing to Asynchronous Backtracking

The performance of $ConcBT$ can be compared to an asynchronous algorithm for solving DisCSPs, Asynchronous BackTracking (ABT) (Yokoo2000). In the ABT algorithm 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 in their mailbox before performing computation.

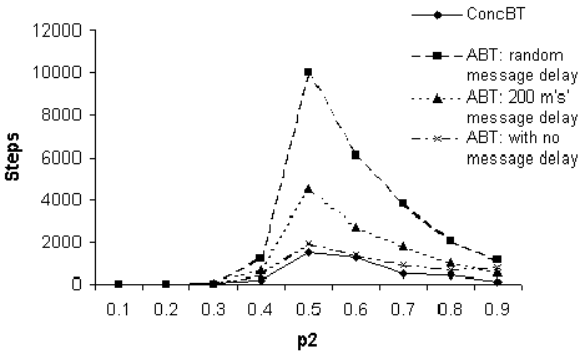

Figure 6: Steps performed by $ConcBT$ and ABT on systems with different patterns of message delays

Figure 6 presents the comparison of $ConcBT$ to ABT, on the same set of DisCSPs with different forms

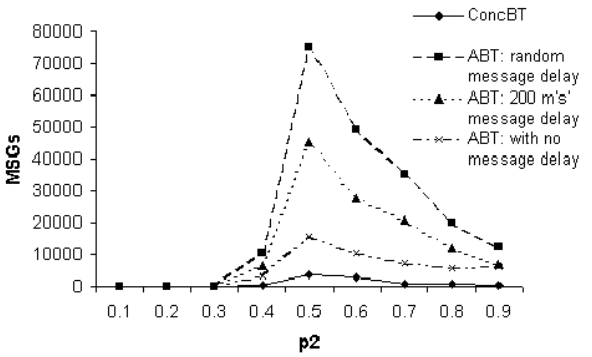

Figure 7: Number of messages sent by *ConcBT* and *ABT* on systems with different patterns of message delay

of communication. *ConcBT* performs fewer steps than *ABT* even on systems with no message delay. When message delivery is not instantaneous, as in real world systems, the performance of distributed search algorithms is changed (Fernandez et. al.2002). When the experiments are performed in a realistic scenario with random message delays, the improvement of *ConcBT* on the harder instances is by a factor of 5. With respect to the number of messages sent, figure 7 shows that *ConcBT* outperforms *ABT* by an even larger factor.

Discussion

A concurrent backtrack search algorithm (ConcBT) with dynamic splitting has been presented. Dynamic splits are triggered by a simple heuristic that counts unsuccessful search steps and splits the search space when a limit is reached. An agent that created a CPA restarts the count each time the CPA returns to it (i.e. backtracks). This policy has the effect of only splitting SPs which delay the search procedure. The experiments in the previous section demonstrate that this simple heuristic is very effective in load balancing. It improves dramatically the efficiency of *ConcBT* with a fixed number of CPAs.

Concurrent BT is a distributed search algorithm that uses a very different mechanism than asynchronous search. A comparison of ConcBT with ABT, on randomly generated DisCSPs produces clear results: Concurrent BT with dynamic re-splitting outperforms asynchronous search. The degree of improvement, in computational effort, of *ConcBT* over *ABT* depends on message delay. When message delays are random the factor of improvement for hard instances of problems is close to 5.

Concurrent backtracking, as proposed in the present paper, 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 (Hamadi2001). There is, however, a major difference between ConcBT and IDIBT (Hamadi2002). The interleaved parallel search algorithm runs multiple processes of an *asynchronous search algorithm* (Hamadi2002). ConcBT runs concurrent *backtrack* search processes and its protocol enables it to perform dynamic splitting of the search space. Our experimental study shows that dynamic splitting of the search space improves the search by a meaningful factor, in contrast to $IDIBT$, where performance deteriorates for splits larger than 2 (Hamadi2002).

References

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

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.

Y. Hamadi. Distributed, Interleaved, Parallel and Cooperative Search in Constraint Satisfaction Networks. In *Proc. IAT-01*, Singappore, 2001.

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

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

N. A. Lynch. Distributed Algorithms. Morgan Kaufmann Series, 1997.

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

Proc. CP-2000 Workshop on Distributed Constraint Satisfaction, Singapore, 22 September, 2000.

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

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

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

M. Yokoo. Distributed Constraint Satisfaction Problems. Springer Verlag, 2000.

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

R. Zivan and A. Meisels *Concurrent Backtrack search on DisCSPs (full version)*, http://www.cs.bgu.ac.il/~zivanr