# Conflicting Agents in Distributed Search

Youssef Hamadi
Publishing Systems and Solutions Laboratory
HP Laboratories Bristol
HPL-2001-222
September 18th , 2001*

E-mail: yh@hplb.hpl.hp.com

distributed constraint satisfaction, distributed AI, search, collaborative software agents,

We extend here the work on interleaved distributed graph based backjumping, IDIBT/GBJ by considering conflicting variables. The resulting method IDIBT/CBJ combines distributed and parallel exploration with more efficient backward phases. We take advantage of our asynchronous framework to implement a more refined and efficient update of detected conflicts. Finally, directed k-consistency is added to our new method which gives IDIBT/CBJ-DkC. We show that this last method extends cooperation in the system.

* Internal Accession Date Only Approved for External Publication

# Conflicting Agents in Distributed Search

Youssef Hamadi
Hewlett-Packards Labs
Filton Road, Stoke Gifford, Bristol BS34 8QZ, United Kingdom
yh@hplb.hpl.hp.com

September 10, 2001

**Abstract**

We extend here the work on interleaved distributed graph based backjumping, IDIBT/GBJ [Ham01, Ham99] by considering conflicting variables [Pro93]. The resulting method IDIBT/CBJ combines distributed and parallel exploration with more efficient backward phases. We take advantage of our asynchronous framework to implement a more refined and efficient update of detected conflicts. Finally, directed k-consistency is added to our new method which gives IDIBT/CBJ-DkC. We show that this last method extends cooperation in the system.

**Keywords: Distributed Constraint Satisfaction, Distributed AI, Collaborative Software Agents, Search**

## 1 Introduction

The constraint satisfaction problem (CSP) is a powerful framework for general problem solving. It involves finding a solution to a constraint network; i.e., finding values for problem variables subject to constraints that are restrictions on which combinations of values are acceptable. This formalism has been extended to tackle distributed problems. In the distributed constraint satisfaction paradigm (DCSP) a problem is distributed between autonomous agents which are cooperating to compute a global solution. The raise in application interoperability combined to the move towards decentralized decision process in complex systems raise the interest for distributed reasoning. In this work we show how to enhance efficiency of distributed search.

The basic method to search for solution in a constraint network is depth-first backtrack search (DFS) [GB65], which performs a systematic exploration of the search tree until it finds an assignment of values to variables that satisfies all the constraints. DFS has been extended to parallel-DFS to speed-up the resolution process [RK93]. Interestingly parallel-DFS showed that under some assumptions, speed-up could be superlinear.

In [Ham01] we have presented *Interleaved Distributed Intelligent BackTracking* an algorithm performing graph-based parallel-DFS in DCSPs. Our algorithm interleaves the exploration of subspaces within each agent. Between distinct agents parallelism is achieved since they can consider distinct subspaces at the same time. Experiments showed that 1) insoluble problems do not greatly degrade performance over a single exploration and 2) on problems with nonuniform search space, IDIBT/GBJ allows superlinear speed-up over a single exploration of a distributed search space.

Here we extend this algorithm to act like the well known sequential conflict-directed backjumping (CBJ) [Pro93]. Interestingly we can take advantage of the asynchronous relation between subproblems by doing a fine and very cost effective update of conflict-sets. Finally, a particular property of CBJ allows the detection of directed k-inconsistencies between subproblems. We add this last feature to our method.

In the following, we first give a basic definition of the CSP/DCSP paradigm, completed by a brief distinction between parallel and distributed search. Then, we present DisAO a distributed variable ordering method, and we describe and analyze IDIBT/CBJ and IDIBT/CBJ-DkC. Afterwards, we give an experimentation with random DCSPs and N-queens problems, followed by a general conclusion.

# 2 Background

## 2.1 Constraint satisfaction problems

A *binary constraint network* involves a set of $n$ *variables* $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D_1, \ldots, D_n\}$ where $D_i$ is the finite set of possible *values* for variable $X_i$ and $\mathcal{C}$ the set of binary constraints $\{C_{ij}, \ldots\}$ where $C_{ij}$ is a constraint between $i$ and $j$. $C_{ij}(a, b) = true$ means that the association value $a$ for $i$ and $b$ for $j$ is allowed. Asking for the value of $C_{ij}(a, b)$ is called a *constraint check*. $G = (\mathcal{X}, \mathcal{C})$ is called the *constraint graph* associated to the network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

A *solution* to a constraint network is an assignment of the variables such that all the constraints are satisfied. The *constraint satisfaction problem* (*CSP*) involves finding a solution in a constraint network.

## 2.2 Distributed constraint satisfaction

A *distributed constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$ is a constraint network (binary in our case), in which variables and constraints are distributed among a set $\{Agent_1, \ldots, Agent_m\}$ of $m$ autonomous sequential processes called *agents*. Each agent $Agent_k$ "owns" a subset $A_k$ of the variables in $\mathcal{X}$ in such a way that $\mathcal{A} = \{A_1, \ldots, A_m\}$ is a partition of $\mathcal{X}$. The domain $D_i$ (resp. $D_j$), the constraint $C_{ij}$ (resp. $C_{ji}$) belongs to the agent owning $X_i$ (resp. $X_j$)[1]. In the present work, we limit our attention to the extreme case, where there are $n$ agents, each only owning one variable, so that $\mathcal{A} = \mathcal{X}$. Thus, in the following, $Agent_i$ will refer to the agent owning variable $X_i$. Of course, the assignment of a single variable can relate the solution of an embedded large subproblem and in fact, each inter-agent constraint can represent a large set of constraints.

Initially, the graph of *acquaintances* in the distributed system matches the constraint graph. So, for an agent $Agent_i$, $\Gamma$ is the set of its acquaintances, namely the set of all the agents $Agent_j$ such that $X_j$ shares a constraint with $X_i$. The *distributed* CSP (*DCSP*) involves finding a solution in a distributed constraint network.

## 2.3 Communication model

For a DCSP, we assume the following communication model [YH96] (in every way classical for distributed systems). Agents communicate by sending messages. An agent can send messages to other agents if and only if it knows their address in the network. The delay in delivering messages is finite. For the transmission between any pair of agents, messages are received in the order in which they are sent. Agents use the following primitives to achieve *message passing* operations:

- $sendMsg(dest, \text{"m"})$ sends message $m$ to the agents in $dest$.
- $getMsg()$ returns the first unread message available.

## 2.4 IDIBT: Distributed and Parallel search

Parallel backtrack search is used to speed-up the resolution process [RK93, Kor81]. Distributed backtrack search faces a situation where the whole problem is not fully accessible; resolution is enforced by collaboration between subproblems.

Both framework use several processing units. In parallel search, N processors concurrently perform backtracking in disjoint parts of a state-space tree. In distributed search, distinct subproblems are spread on several processing units and backtracking is performed by the way of collaboration.

Part a) of figure 1 presents an example of parallel exploration. Here, the problem is duplicated on two processors $P_0$ and $P_1$. $P_0$ is in charge of the subspace characterized by $X_1 = a$, $P_1$ explores the remaining space. During the computation, message passing is useless. However, since a processor can exhaust its task before another (good heuristic functions, filtering, ...), dynamic load balancing is used [RK93]. Usually, an idle unit asks a busy one for a part of its remaining exploration task.

[1] We suppose that the constraint network is such that $(\mathcal{X}, \mathcal{C})$ is a symmetric graph.

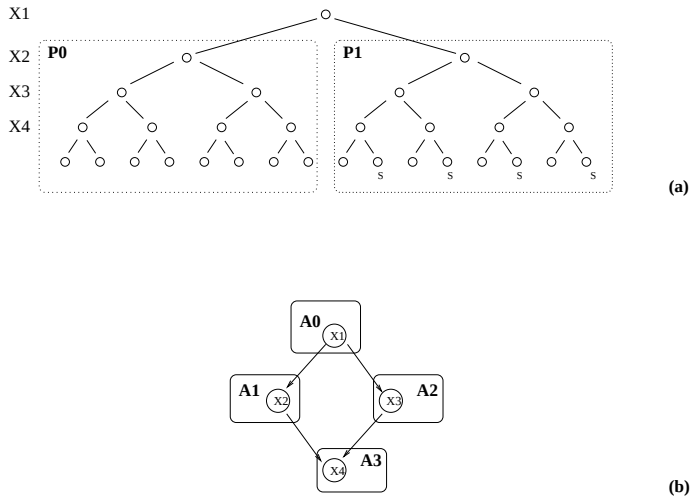


Figure 1: **Tree searches:** (a) parallel search, (b) distributed search

Part b) of the figure presents a distribution of this 4-variables problem between four autonomous agents. Here, state-space exploration uses local resolution for each subproblem with negotiation on the shared constraints.

To add parallel search in our distributed framework, we must divide a search space in independent parts. In each part a distributed conflict-directed backtrack search will take place. In the system, we will have two kind of agent with distinct behaviors.

- a *Source* agent, which will partition its search space in several subspaces called *Context*
- the remaining agents which will try to instantiate in each context.

There is no duplication of processing units here. Agents will successively consider search in the different contexts. This interleaving will be achieved by message passing operations. The context of resolution added within each message will allow an agent to successively explore the disjoint search spaces. Since distinct agents can simultaneously consider and operate in different context, IDIBT realizes a parallel exploration of the search space too. In [Ham01, Ham99], we showed that 1) insoluble problems do not greatly degrade performance over a single-context exploration and 2) superlinear speed-up can be achieved when the distribution of solution is nonuniform.

# 3 Conflicting Agents in Distributed Search

Before presenting our new methods, we recall here the different features of the IDIBT algorithm.

IDIBT mixes parallel and distributed search. It realizes a DFS between the agents of a distributed CSP. DFS is a general complete resolution technique widely used for its storage efficiency. Given a variable and value ordering, it generates successive assignments of the problem variables. It tries to extend a partial assignment by taking the next variable in the ordering and by assigning it a value consistent with previously assigned variables. If no value can be found for the considered variable, the algorithm backtracks.

IDIBT/GBJ use the principles of Dechter's GBJ [Dec90]. When an assignment cannot be extended, it goes back to the nearest connected agent in the system. With our IDIBT/CBJ scheme, this behavior will be more refined. Each agent will only consider conflicting agents when backtracking. By this way, it will always address a subset of connected agents.

Our framework is totally asynchronous but we need an ordering between related agents to apply the backtracking scheme which ensures completeness. In the following we present our distributed ordering method followed by the IDIBT search process.

## 3.1 Distributed Agent Ordering

The practical complexity of a search process is highly dependent on user's heuristic choices such as value/variable ordering. Usually these heuristics take advantage of domain-dependent knowledges. Each agent can use particular heuristics in the exploration of its subproblem. But in the DCSP, agents must collaborate to use an efficient ordering in the distributed search process. We present here DisAO, a generic method for a distributed computation of any static agent ordering. With this algorithm, agents cooperatively build a global ordering between the subproblems. This ordering defines a hierarchical relation between the agents.

### 3.1.1 Algorithm

In our system, each agent locally computes its position in the ordering according to the chosen heuristic. Concretely, each agent determines the sets $\Gamma^+$ and $\Gamma^-$, respectively *children* and *parent* acquaintances, w.r.t. an evaluation function $f$ and a comparison operator *op* which totally define the heuristic chosen. This is done in the lines 1 to 2 of Algorithm 1. Notice that the evaluation function $f$ can involve some communication between the agents. To avoid a complex communication behavior, it is better to use heuristics for which the associated function $f$ involves only local communications between neighbor agents.

**Algorithm 1: Distributed variable ordering**

```
   begin
       % Γ split;
1      Γ+ ← ∅; Γ− ← ∅;
       for each Agent_j ∈ Γ do
           if (f(Agent_j) op f(self)) then Γ+ ← Γ+ ∪ {Agent_j};
2          else Γ− ← Γ− ∪ {Agent_j};
       % Γ− ordering;
3      max ← 0;
       for (i = 0; i < |Γ+|; i++) do
           m ← getMsg();
           if (m = value:v; from:j) then
               if (max < v) then max ← v;
       max++;
       sendMsg(Γ−, "value:max; from:self");
       sendMsg(Γ+, "position:max; from:self");
       for (i = 0; i < |Γ−|; i++) do
           m ← getMsg();
           if (m = position:p; from:j) then Level[j] ← p;
       Order Γ− according to Level[] ;
4      Extend Γ− ;
   end
```

After that, agents know their *children* ($\Gamma^+$) and *parents* ($\Gamma^-$) acquaintances. During the search, they will send assignment value to children, and in case of dead-end, they will backtrack to the *first* agent in $\Gamma^-$. So, we need a total ordering on $\Gamma^-$. This is done in the second part of Algorithm 1 (lines 3 to 4). Agents without children state that they are at level one, and they communicate this information to their acquaintances. Other agents take the maximum level value received from children, add one to this value, and send this information to their acquaintances. Now, with this new environmental information, each agent rearranges (total order) the agents in its local $\Gamma^-$ set by increasing level. Ties are broken with agent tags. Finally, for fitting each total order $\Gamma^-$, the constraint graph is extended with zero or more additional edges (lines 4). These new edges are tautological constraints. Their purpose is the enforcement of completeness by local search space initialization in the forward exploration phases (see section 3.4). We do not present details about this computation here. In summary, each agent communicates its ordered $\Gamma^-$ set to its parents. These agents can locally modify their sets by adding lower (resp. higher) agents in their $\Gamma^-$ (resp. $\Gamma^+$). This process is repeated until stabilization; i.e., no more $\Gamma$ modification.

Figure 2 gives an illustration of this distributed processing for the *max-degree* variable ordering heuristic. On the left side of the figure a constraint graph is represented. For achieving the max-degree heuristic, Algorithm 1 must be called by each agent with the function $f(Agent_i) = |\Gamma_i|$ (where $\Gamma_i$ is the set of

acquaintances of $Agent_i$) and the comparison operator $op = '<'$. In case of ties, this operator can break them with agent tags.

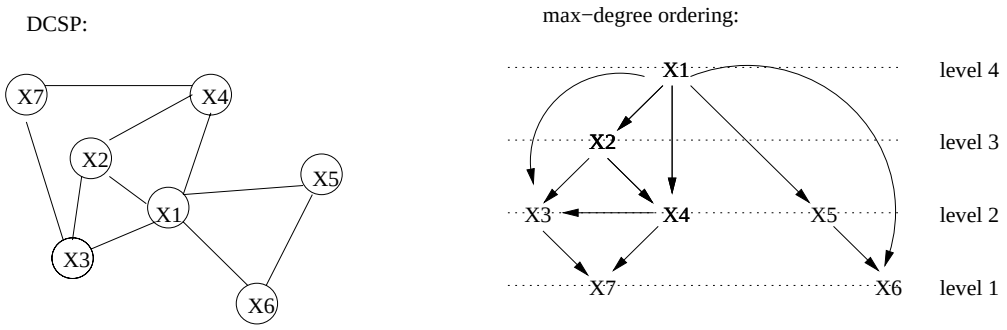


Figure 2: Distributed variable ordering

Once Algorithm 1 has been applied, the static variable ordering obtained is the one presented on the right side of Fig. 2. Arrows follow the ordering relation, which represents the assignment transmission order of the search procedure. The link between $Agent_4$ and $Agent_3$ comes from the interconnection of $Agent_7$'s parents. $Agent_7$ will go back to $X_3$ then to $X_4$ if $Agent_3$ has no remaining solution. During forward exploration, a change in $X_4$ will be reported to $X_3$ and to $X_7$. These agents will then get back their whole search space.

### 3.1.2 Analysis

In the worst case, w.r.t. a fully connected network with $n$ agents, the split of $\Gamma$ uses $O(n)$. The exchange of **value** among the path of $n$ agents use $O(n^2)$ messages; i.e., level one agent sends $n-1$ messages, level 2 agent $n-2$ and so on. These messages can overlap, this bring $O(n)$ local operations for performing these transmissions. The transmission of **position** messages is similar but from the top to the bottom. The extension of the ordering in the hierarchy adds no link but requires $O(n^2)$ message to exchange $\Gamma^-$ sets. According to that, DisAO uses $O(n)$ local operations and $O(n^2)$ messages in the worst case[2].

PROPERTY 3.1 ()
$\forall A_i$, if $\exists A_j, A_k$ such that $A_j \rightarrow A_i$ and $A_k \rightarrow A_i$, then $\exists A_j \rightarrow A_k$ or $\exists A_k \rightarrow A_j$.

We have $A_j \rightarrow A_1, A_i$ and $A_k \rightarrow A_2, A_i$ with $A_1 \in \Gamma^-(A_i)$ and $A_2 \in \Gamma^-(A_i)$. By definition we have $f(A_1) opf(A_2)$ or $f(A_2) opf(A_1)$ then by $\Gamma^-$ extension we have $A_2 \rightarrow A_1$ or $A_1 \rightarrow A_2$. We can follow the previous reasoning by considering $A_i = A_1$ or $A_i = A_2$.

PROPERTY 3.2 ()
For a problem $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$, if $(X, C)$ is connected, the directed graph computed with DisAO has an unique agent such that $\Gamma^- = \emptyset$.

The proof is straight forward, if we consider 3.1. In a DisAO ordering, there is a unique source and the hierarchy is made of subproblems (involving several agents) organized in a global tree.

Finally, we can remark that in the resulting ordering, at a particular level, unconnected agents are independent. Connected ones are linked by tautological constraints. This means that their information will just initialize the search space without loosing current assignment. Hence, in each level, agents can perform parallel computations at the same time. This observation will be important when we will consider the complexity of distributed search.

[2] Of course for predefined applications, the DisAO pre-processing step could be avoided.

## 3.2 IDIBT/CBJ: Using Conflicts Between Agents

While jumping back, Dechter's GBJ is directed by topology. With IDIBT, we have the same behavior with the only difference that sometime topology is extended to join asynchronous behavior and completeness. Prosser's CBJ is directed by conflicts. This sequential algorithm stores with each variable $i$ a "conflict-set" which keeps the subset of the past variables in conflict with some assignment of $i$. When a dead end occurs, CBJ jumps back to the deepest variable $h$ in conflict with $i$. If a new dead end occurs, it jumps back to $g$, the deepest variable in conflict with neither $h$ or $i$. Finally, each time CBJ jumps back from $i$ to $h$, the variables $j$ such that $h < j \leq i$ get back their search space and an empty conflict-set.

In the following we detail the adaptation of this behaviour in a distributed framework. Like we did before with the adaptation of Dechter's GBJ, our algorithm enhances the sequential CBJ scheme by saving the maximum of previous work between agents.

### 3.2.1 Principles

Each agent will maintain a conflict-set which will be used during backjumping. But here, during a jump from $i$ to a conflicting agent $h$ our distributed framework can easily preserve previous work. While CBJ proactively reinitialize the search space and conflict-set of each variable $j$ such that $h < j \leq i$, IDIBT/CBJ will do nothing. In fact, these updates are reduced and delayed as follow.

Our strategy is that when a value is addressed to $i$ by an agent $j$, the receiver can selectively prune its conflict-set to keep conflicts unrelated with the new information from $j$. The elements of the local conflict-set are ordered ($<_o$) thanks to DisAO (see section 3.1). These positions are computed from the bottom to the top. For each conflicting agent $h$, we have two configurations:

- $j <_o h$, the conflict with $h$ is still valid. According to that the corresponding pruning in the local search space can be kept. Indeed, $h$ is located in a position higher than the location of the sender. According to that, the new value for $j$ cannot cancel previous decisions about the current search space (remaining values and conflict-set). This argument is correct if we consider the following. A jump back initially occur to the deepest variable in conflict, here that mean the highest agent in the ordering. So conflicts detected from $h$ are unrelated to $j$ since $h$ is always checked before $j$ when the agent tries to get an assignment.

- $h <_o j$, here the deletion of values raised by h's value are not independent from j's values. The local search space can recover these values and $h$ is removed from the conflict-set.

To implement the previous, each agent records for each member of the conflict-set the corresponding pruning in the local search space.

This conservative behavior generalizes the observation made by Prosser in [Pro93] for the sequential CBJ. In this paper the author remarks that during the reinitialization phase of its procedure (following a backjump from $i$ to $h$), it may be the case that max-list(conf-set[$j$])$< h$, in which case we can continue to believe the current pruning for $j$. In the sequential framework, we would have to examine all future conflict sets whenever backjumping takes place. This could be expensive. Here the asynchronism in the system allow us to implement efficiently a generalization of the previous. Of course such generalization could be made for sequential CBJ[3].

This improvement makes IDIBT/CBJ close to backmarking [Gas77, LHB94, Ham99] and in a limited way close to dynamic backtracking [Gin93]. The previous behavior could be enhanced by cutting dependencies between variables during the detection of conflicts. Nevertheless, this behavior should be costly since the pruning made according to a particular acquaintance could not be used for successive tests.

### 3.2.2 Algorithm

The global scheme of the search process is the following (see algorithm 2 and data structure below). In the initialisation phase (lines 1 to 3), the source agent divides its search space in $NC$ subspaces. Remaining agents will use the same space $D$ in each context. In each context $c$ each agent initializes its conflict-set and

[3] Personnal communication to P. Prosser, summer 1998.

assign its variable. Each timestamp counter $valueCpt_c$ is then set to one. After that, each agent informs its children of its chosen value (message content starting by "**infoVal**").

Interactions start at line 4. Here each incoming message is interpreted in a particular context $c$ (lines 5 and 6).

An "**infoVal**" message from acquaintance $j$ is processed as follow (line 7). First the reported value is stored in $value[j]_c$ then the associated timestamp $valueCpt[j]_c$ is incremented. Finally the agent tries to get a value compatible with the new message. If a compatible value is found, an "**infoVal**" message with context $c$ informs children of the new choice[4]. If no value satisfies the constraints with the agents in $\Gamma^-$, a backtrack message is sent in context $c$ to the nearest conflicting agent (message content starting by "**btSet**" in line 8). This message includes the computed conflict-set and beliefs about the timestamps of its members $valueCpt[conflictSet_c]_c$.

The receiver of the backtrack message (line 9), checks its validity by comparing its timestamp with the reported one and by checking that shared acquaintances are reported with the same timestamps too (function $contextConsistency$). In case of different values, this means that the sender and/or the receiver have not yet received some information. Backtrack decision could then be obsolete or badly interpreted.

When the comparison matches, reported conflicts are merged with the local ones. The current value $myValue_c$ is added to the pruned values for the included agents; i.e., the corresponding pruning in the local search space. Then, if the agent can find a compatible $myValue_c$ in the remaining search space, this value is addressed to children in line 10. If such a value cannot be found, we must consider two cases. The first one is an agent without possibility for backtracking (empty conflict-set, line 11). This agent has detected problem insolubility in the subspace $c$. A message $noSolution$ in context $c$ is sent to a *System* agent. This extra agent stops the distributed computation in context $c$ by broadcasting a *stop* message in the whole multi-agent system. With this information agents can stop the processing of context $c$ messages. If **all** the context have no solution, the computation is finished. In addition, it can also stop the computation when a solution is found. A global state detection algorithm [CL85] is used to detect whole satisfaction. Global satisfaction occurs when in **a** particular context $c$, agents instantiated according to parent constraints are waiting for a message (line 5) and when no message with context $c$ transits in the communication network.

If there exists a conflicting agent for backtracking, the agent address a backtrack message to the nearest agent in the augmented conflict-set (line 12). Now the sender is waiting for an incoming message, but if the goal is to maximize satisfaction in the system, the agent could get back its initial local search space and a value compatible with previous ones. This simple addition is a simple way toward a max-sat strategy.

**Primitives and data structures**

IDIBT/CBJ uses the following structures and methods:

- $NC$ is the number of contexts. self is the agent running the algorithm, $D_{self,c}$ is its domain in context $c$.
- $myValue_c$ current value in the context $c$. $myCpt_c$ current instantiation number in context $c$. This value will be used as a timestamp in the system.
- $value[]_c$ stores parent acquaintances values in context $c$. $valueCpt[]_c$ stores for each parent the current instantiation number, in the right context.
- $conflictSet_c$, this set records the conflicting acquaintances in context $c$.
- $updateSpace(j, c)$, according to an *infoVal* message from $j$, implements in context $c$ the selective update of the search space described above.
- $getValue(type, c)$, returns the first value in $D_{self,c}$ compatible with agents in $\Gamma^-$, starting at $myValue_c$[5]. During this search, $conflictSet_c$ is updated. If a compatible value is found, $myCpt_c$ is incremented.

[4] Of course, current value $myValue_c$ can already satisfy the constraints with $j$, in which case, information of children is useless.

[5] The search for a new compatible value starts from the current value for keeping the maximum of previous work. For ensuring completeness, the values added by *updateSpace* that are before $myValue_c$ in $D_{self,c}$ are put at the end of $D_{self,c}$.

- first($S$) returns the first element of an ordered set $S$. With our application, returns the nearest agent in $S$. *merge*(s1,s2) takes two ordered sets and returns their ordered union.
- *contextConsistency*($set, reportedValueCpt, c$), $set$ contains an ordered list of agents, $reportedValueCpt$ contains for each agent in $set$ timestamps computed by the sender of the current message. This function ensures that, firstly reported timestamp for self is the good one; i.e., equal to $myCpt_c$, secondly that for the shared acquaintances agents, reported timestamps are the same than in $valueCpt[]_c$. This mechanism ensures that agents have the same beliefs about the shared parts of the system.
- The previous $sendMsg$ function becomes $sendMsg(dest, m, c)$, which sends message $m$ to the agents in $dest$ in context $c$.

**Algorithm 2: IDIBT/CBJ**

```
   begin
1    if (Γ⁻ = ∅) then Split domain D in D_self,1 .. D_self,NC;
     for (1 ≤ c ≤ NC) do
       if (Γ⁻ != ∅) then D_self,c ← D;
       conflictSet_c ← ∅;
2      myValue_c ← getValue(info, c);
       myCpt_c ← 1;
       sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
3      end_c ← false;
4    while (∃c|end_c = false) do
5      m ← getMsg();
6      c ← m.context;
       if (m = stop) then end_c ← true;
7      if (m = infoVal:a; from:j) then
         value[j]_c ← a;
         valueCpt[j]_c + +;
         updateSpace(j,c);
         myValue_c ← getValue(c);
         if (myValue_c) then
           sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
         else
8          sendMsg(first(conflictSet_c), "btSet:conflictSet_c; Values:valueCpt[conflictSet_c]_c", c);
9      if (m = btSet:set; Values:reportedValueCpt) then
         if (contextConsistency(set, reportedValueCpt, c)) then
           conflictSet_c ← merge(set, conflictSet_c);
           myValue_c ← getValue(c);
           if (myValue_c) then
10           sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
           else
11           if (conflictSet_c = ∅) then
               sendMsg(system, "noSolution", c);
               end_c ← true;
             else
12             sendMsg(first(conflictSet_c), "btSet:conflictSet_c; Values:valueCpt[conflictSet_c]_c", c)
   end
```

## 3.3 Directed k-consistency

CBJ has been extended to achieve directed k-consistency [Pro93]. We can extend our algorithm to IDIBT/CBJ-DkC as follow. This will allow us to present an example of cooperation between several search processes.

During backjumping, when an agent $i$ address a conflict-set $\{h = v(h)\}$ to $h$, that mean that the value $v(h)$ is incompatible with any value of the domain of $i$. This value is arc-inconsistent and can be definitively

removed from the domain of $h$. Here the asynchronism can make the backjump obsolete; i.e., $h$ has a new value. But this information is still useful and $v(h)$ must be removed[6].

To illustrate directed k-inconsistency, consider a configuration where an agent $j$ exhausts its domain and constructs an ordered conflict-set $\{i = v(i), h = v(h)\}$. That mean that each value $v(j)$ is inconsistent with $v(h)$ or $v(i)$. Consider now that $i$ has no conflict with its $\Gamma^-$. Now, assume that each new value addressed from $i$ to $j$ brings a new backjump to $i$. At the end, $i$ exhausts its search space too and backjump to $h$ which can definitively remove $v(h)$ from its domain.

Within IDIBT several DFS search are parallelized (see section 2.4). Each agent interleaves explorations and each k-inconsistency detected in a particular context can benefit to other contexts. This sharing makes IDIBT very close to cooperative frameworks [HH93, HW93]. But while classical cooperative frameworks have to suffer from overheads to communicate useful informations, here the interleaving within each agent makes sharing very efficient.

To implement the previous within IDIBT, we have to add the following line

9.1 $if(|set| = 1)\ remove(D_{self}, set(self))$

where *set(self)* represents the reported value.

## 3.4 Analysis

### Completeness

PROPERTY 3.3 ()
When an agent $A_i$ changes its instantiation, agents $A_j$ such that $\exists A_i \rightarrow A_j$ will reconsider their whole search space.

The proof is direct if we consider the algorithm 2. When an agent changes its value, $\Gamma^+$ agents receive it. These agents can keep their current instantiation or change it, but they always resume their local search space since the conflict-set is deleted from the sender location to the nearest agent. By propagation of instantiations between agents, 3.3 is verified.

PROPERTY 3.4 ()
If $A_i$ changes its instantiation according to a **btSet** message initially upcoming from $A_j$, each agent $A_k$ included in the conflict-set of agent $A_j$ has exhausted its search space.

This last property is obvious. Properties 3.1, 3.3 and 3.4 ensure completeness of the exploration. They prove that according to the DisAO computed ordering, backjumping between agents is made in an exhaustive way.

### Termination/Correctness

Termination is ensured by search exhaustivity and by the fact that DisAO orders are acyclic. The use of a state detection algorithm [CL85] which stops the system when any context $c$ is stuck on a solution gives correctness. Interestingly the use of several context within IDIBT do not significantly change the overhead brought by the Chandy's method. In fact it is easy to generalize the method to manage the monitoring of the different context without raising the message passing overhead; i.e., each monitoring message includes the status of the different contexts.

### Complexity

Search complexity is exponential in the number of variables. But in a distributed execution, rooms are open to use the relative independence between subproblems. This can enhance complexity results. In the following, $level_j$ represents the set of agents with a computed level $j$ and $h$ the highest level in the ordering.

[6] Since IDIBT transmits timestamps instead of beliefs about assigned values, the sender adds the value which has to be deleted each time it address a singleton-conflict-set.

DEFINITION 3.1 ()
A DisAO ordering is called *additive* if $\forall b \in level_j \mid 1 \leq j \leq h$, $\nexists$ agents $a, a' \in level_i$ with $1 < i \leq h \mid a \rightarrow b$ and $a' \rightarrow b$.

THEOREM 3.1 ()
A DCSP $P$ with domain sizes $d$, using an additive DisAO ordering has a worst case time complexity,

$$O(\prod_{l=1}^{h} |level_l| \times d)$$

To prove that we must remark that with an additive ordering, during backtracking, the union of two $\Gamma^-$ set do not include two agents at the same level. Then a backtracking occurs between distinct level and at each time considers at most $d$ values. The whole problem is solved by considering at each level combinations of values. Since at each level, agents are independent, the number of possibilities is made by the sum of domains size.

When the ordering produced by DisAO is not additive, the complexity of a backtracking depends on the size of the longest path between agents. In the worst case we have an $O(d^n)$ complexity. We must remark here that DisAO was not made to construct additive hierarchies; its purpose was to add more parallelism by extracting subproblems independence. Nevertheless, we think that it must be possible to embolden parallelism while maximizing the additive property.

# 4 Conclusion

This work extends previous work on graph based backjumping. It detects conflicts between agents. Interestingly the distributed framework allows us to make efficient update of conflicts. The work was extended to achieve directed-k-consistency. With this last version, our algorithm extends the cooperation. In multi-context search, dkC can benefit to disjoint search spaces. We are currently doing experiments with all these methods. First results are encouraging.

# References


[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, Feb 1985.

[Dec90] R. Dechter. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *AI*, 41(3):273–312, 1990.

[Gas77] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *IJCAI*, volume 1, page 457, 77.

[GB65] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 65.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *JAIR*, 1:25–46, 1993.

[Ham99] Y. Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués.* PhD thesis, Université Montpellier II, 1999. (in french).

[Ham01] Y. Hamadi. Interleaved backtracking in distributed constraint networks. In IEEE, editor, *13th International Conference on Tools with Artificial Intelligence*, page (to appear), 2001.

[HH93] T. Hogg and B. A. Huberman. Better than the best: The power of cooperation. In Lynn Nadel and Daniel Stein, editors, *1992 Lectures in Complex Systems*, volume V of *SFI Studies in the Sciences of Complexity*, pages 165–184. Addison-Wesley, Reading, MA, 1993.

[HW93] T. Hogg and C. P. Williams. Solving the really hard problems with cooperative search. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 231–236, Menlo Park, CA, USA, July 1993. AAAI Press.

[Kor81] W. Kornfeld. The use of parallelism to implement a heuristic search. In Patrick J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 575–580, Los Altos, CA, 24–28 August 1981. William Kaufmann.

[LHB94] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Strategies for distributed constraint satisfaction problems. In *Proc. of the 13th Int. Workshop on DAI*, pages 207–221, Jul 1994.

[MPI94] Message Passing Interface Forum MPIF. MPI: A message-passing interface standard. *Int. Journal of Supercomputer Applications*, 8(3/4), 1994.

[Pro93] P. Prosser. Domain filtering can degrade intelligent backjumping search. In *IJCAI*, pages 262–267, 1993.

[RK93] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, Apr 1993.

[YH96] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *ICMAS*, pages 401–408, Dec 1996.