Maintaining Hierarchical Distributed Consistency

Marius Călin Silaghi and **Djamila Sam-Haroud** and **Boi Faltings**
Swiss Federal Institute of Technology, Lausanne
{silaghi,haroud,faltings}@lia.di.epfl.ch

ABSTRACT

Recent work on bringing classical CSP techniques to distributed environments has focused either on backtrack search or on local consistency. We consider the "natural" issue of combining the two techniques. A new distributed algorithm, called MHDC (Maintaining Hierarchical Distributed Consistency), is presented which incorporates distributed consistency into asynchronous backtracking. One of its main characteristics is to consider consistency maintenance as a hierarchical task. Enforcing the hierarchies of consistency and performing search can then be done with a high degree of asynchronism. This gives the agents more flexibility and freedom in the way they can contribute to search, and increases parallelism. As expected, the experimental results show that substantial gains in computational power can result from combining distributed search and distributed local consistency algorithms.

1. INTRODUCTION

Constraint satisfaction has proven to be a highly successful paradigm for solving combinatorial problems such as resource allocation, scheduling or planning. A constraint satisfaction problem (CSP) is classically defined as a set of variables taking their values in particular domains and subject to constraints which specify consistent value combinations. Solving a CSP amounts to assigning to the variables, values from their domains, so that all the constraints are satisfied.

Distributed CSPs arise when information about variables and/or constraints is distributed among different agents. They provide a natural framework to deal with the increasingly diverse range of distributed real world problems emerging from the fast evolution of communication technologies.

In a centralized setting, backtrack search is the principal mechanism for solving CSPs. It is commonly combined with local consistency techniques to limit the combinatorial explosion. These techniques reduce the size of the search space by removing local inconsistencies. Maintaining Arc Consistency (MAC) [8], for example, is one of the most powerful algorithms for solving hard CSPs. It combines arc-consistency with backtracking.

In the last years many authors have been working on bringing centralized CSP techniques to the distributed framework. As a result, we now have at disposal several asynchronous backtracking [15, 4, 10] and distributed arc-consistency algorithms [18, 3]. Nevertheless, no work has yet been reported on the combination of distributed search and distributed local consistency techniques.

We tackle this issue and propose a new MAC like algorithm called Maintaining Hierarchical Distributed Consistency (MHDC). MHDC can incorporate either distributed bound- or arc-consistency[1]. Without loss of generality, we will only concentrate on bound-consistency in the rest of this paper. Bound-consistency enforces arc-consistency on the outer bounds of the variables domains only. It requires the domains to be ordered, but spares a significant amount of work and space [12]. We propose a hierarchical handling of bound- and arc-consistency which enables the instantiation and consistency maintenance steps to be performed asynchronously. The preliminary experimental evaluation shows that MHDC significantly increases the computational power of asynchronous backtracking, similarly to its centralized counterparts.

2. BACKGROUND & DEFINITIONS

Bound-consistency is a relaxation of arc-consistency originally proposed for continuous domains. It is integrated in the most successful techniques for numerical CSPs. A centralized algorithm for maintaining bound-consistency on discrete problems is presented in [12] and has shown much potential. Bound-consistency enforces arc-consistency on the outer bounds of ordered domains.

In this work we propose to integrate distributed bound-consistency in Asynchronous Aggregation Search (AAS) [10], a recent distributed search technique with the following characteristics:

- Constraint-orientation: In the common definition of DCSPs [16], variables are distributed among agents so that each variable can only be assigned values by a single agent. In that setting, constraints may need to be revealed to any agent that controls a variable in the corresponding constraint. AAS considers the dual case where constraints are private (i.e. each constraint may be controlled by a single agent) but any agent can propose to change any of its variable. This paradigm is intended to better respond to the needs of certain real world situations, such as negotiation, where the variables are public, but the constraints private.
- Aggregation-based instantiations: In AAS the agents exchange messages not about assignments to individual variables, but about ranges of values for combinations of variables (aggregates). This allows for obtaining efficiency gains over the existing asynchronous search algorithms.
- Bounded nogood recording: There exists three AAS algorithms [10]: AAS2 which is based on full nogood recording similarly to the asynchronous search algorithm of [16] and AAS1 and AAS0 which store a bounded number of nogoods. AAS1 proceeds similarly to dynamic backtracking by eliminating the nogoods which depend on a modified instantiation. AAS0 is a novel algorithm which modifies AAS1 in such a way that each agent only maintains a single nogood. AAS0 merges the nogoods AAS1 would generate into a single nogood using a relaxation rule. In fact, in AAS, the policy for storing nogoods is flexible and as many nogoods as wanted can be added to those maintained by AAS0 or AAS1.[2]
- Generality: AAS generalizes the existing asynchronous backtracking algorithms since it can reproduce the behavior of both ABT [16] and DIBT [4][3] under particular conditions.

[1]Distributed arc-consistency is defined as in [18], where the local CSP of each agent is seen as one constraint.

[2]Criteria for selecting nogoods with higher relevance (e.g. the number of inferences) can be used. Other useful criteria appear in [2, 6]

[3]DIBT is obtained in AAS0 when agents have constraints as if con-

We mention that an integration of approximation techniques into Asynchronous Search (ABT) has also been described in [5]. A synchronous approach close to AAS2 has been proposed for solving design problems (see [1]).

A DCSP is composed by a set of n agents where each agent has a set of requirements. The requirements of an agent A^i is modeled by a local CSP denoted by $CSP(A^i)$. The variables, respectively the constraints of $CSP(A^i)$ are called the *local constraints*, respectively *local variables* of the agent A^i.

A solution to the DCSP must satisfy the union of the local CSPs of all its agents. The union of two CSPs P_1 and P_2 is defined as: $P_1 \cup P_2 = (V_1 \cup V_2, C_1 \cup C_2, D_1 \cup D_2)$, where the common variables in V_1 and V_2 should have identical domains in D_1, respectively D_2[4].

MHDC is implemented on a distributed platform which provides a Broker's interface and capability as well as a configurable dispatcher queue for distributing the messages received by the agents. This platform is implemented in sJavap [9], an extension of Synchronous Java which allows the usual sJava primitives to associate priorities to concurrent transitions.

3. DISTRIBUTED BOUND-CONSISTENCY

We start by presenting succinctly the distributed bound-consistency algorithm that we combine with AAS. This algorithm, called DHC, builds on AC3 [7] and is similar to DAC [18].

In DHC, each agent A^i maintains a revision queue Q_i and a list of labels L_i. Q_i stores a set of constraints involved in $CSP(A^i)$ and initially contains all the local constraints of A^i. The basic form of a label is a pair (`var`,`dom`) which associates a locally bound-consistent domain `dom` to any variable `var` of $CSP(A^i)$. The agents communicate by sending `propagate` messages. The argument of a `propagate` message is a list of labels.

Each agent A^i starts by enforcing bound-consistency on its own local CSP, $CSP(A^i)$. This computation initializes the labels of L_i which are then sent to all the agents interested in the same variables. When A^i receives a new label (`var`,`new-dom`) via a `propagate` message, it combines this label with the corresponding entry (`var`,`old-dom`) of L_i. Such a combination is done by domain intersection and if it results in a reduction of `old-dom`, all the constraints of $CSP(A^i)$ involving `var` are reinserted in Q_i. Q_i is then used for re-enforcing bound-consistency on $CSP(A^i)$. The resulting modified labels, if any, are further broadcast to the concerned agents until convergence is reached. The convergence is signaled by silence on the network when no agent works. The convergence of DHC is a direct consequence of the facts that the domains are finite and that a message is sent only when a label is reduced. Following the complexity of AC3, the maximum number of generated messages is a^2nd and the maximum number of sequential messages is nd (n:number of variables, d:maximum domain size, a:number of agents). There can be a^2 simultaneous messages.

4. MAINTAINING DISTRIBUTED BOUND-CONSISTENCY

In the centralized framework, the MAC and MHC algorithms maintain arc-, respectively bound-consistency, after each variable, respectively constraint, is assigned a value [8, 13, 12]. A simple adaptation of these algorithms to the distributed environment would require the consistency maintenance and instantiation steps

trolling one variable (minimal/no aggregation), are ordered according to a DFS criteria [17], and each of them knows the tautology $\models$ instantiations-of-ancestors$\rightarrow \emptyset$ whose premise is an absorbant for all incoming nogoods.

[4]A future extension may consider the case of intersecting the eventual different domains, or dealing with "values" forgotten or missed by some agents.

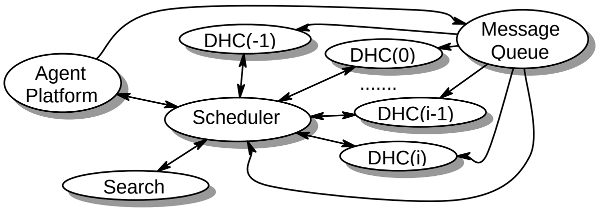

Figure 1: *The architecture of an agent A^i: active objects (threads) deal with each level of DHC. In the idle time, the Scheduler thread performs backtrack search.*

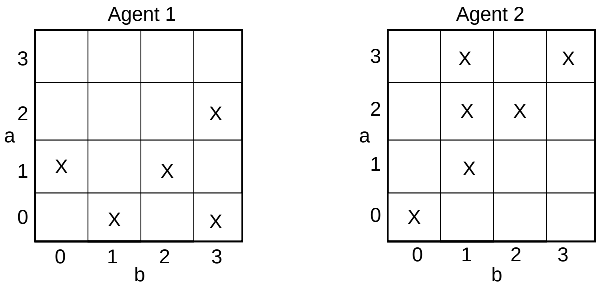

Figure 2: *A simple problem where running search and consistency maintenance asynchronously helps.*

to alternate sequentially. This not only restricts parallelism, but also requires a heavy machinery of termination tests. (The convergence of each local propagation needs to be detected). Such a technique will be referred to as *synchronous MDC* in the rest of the paper.

In order to increase the efficiency of the overall process, we propose to allow some degree of asynchronism between consistency maintenance and search. To achieve this goal, we consider bound-consistency maintenance as a hierarchical task.

We define $\mathtt{DHC}^{\mathtt{i}}_{-1}$ as the particular DHC process that can be launched by agent A^i when it has no information about the instantiation of other agents. $\mathtt{DHC}^{\mathtt{i}}_{-1}$ corresponds to the elementary level where the original global CSP is brought bound-consistent.

We also define a label of level k as a label computed by (bound) consistency using the instantiation of agent A^k. We then define $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{k}}$, with $i \geq k \geq 0$, as the DHC process of level k that can be launched by A^i. $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{k}}$ is intended to compute bound-consistent labels of level k for the CSP:

$$CSP(A^i) \cup \{ \bigcup_{-1 \leq j \leq k} Label(j) \} \cup \{ \bigcup_{k<j<n, j \neq i} CSP(A^j) \}$$

where $Label(j)$ is a CSP constraining any solution to be contained in the labels of level j of all agents.

The basic idea then consists of enabling the $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{j}}$'s, with $-1 \leq j \leq i-1$, to run asynchronously in A^i for different j's, together with the asynchronous backtracking of AAS (see Figure 1).

In a *synchronous MDC* each agent A^i would only launch the process $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{i}}$ when the $\mathtt{DHC}^{\mathtt{k}}_{\mathtt{j}}$ have reached convergence for all k and for all $j < i$.

Our motivation is that although maintaining full bound-consistency is likely to reduce significantly the width of search in general, cases might occur where it is either useless, more expensive than search or simply not beneficial for the direct next branch of search. In effect, search alone, or combined with partial forms of bound-consistency can sometimes provide better alternatives, and parts of this consistency do not need to be waited for. On the other hand, the reductions obtained through consistency on a

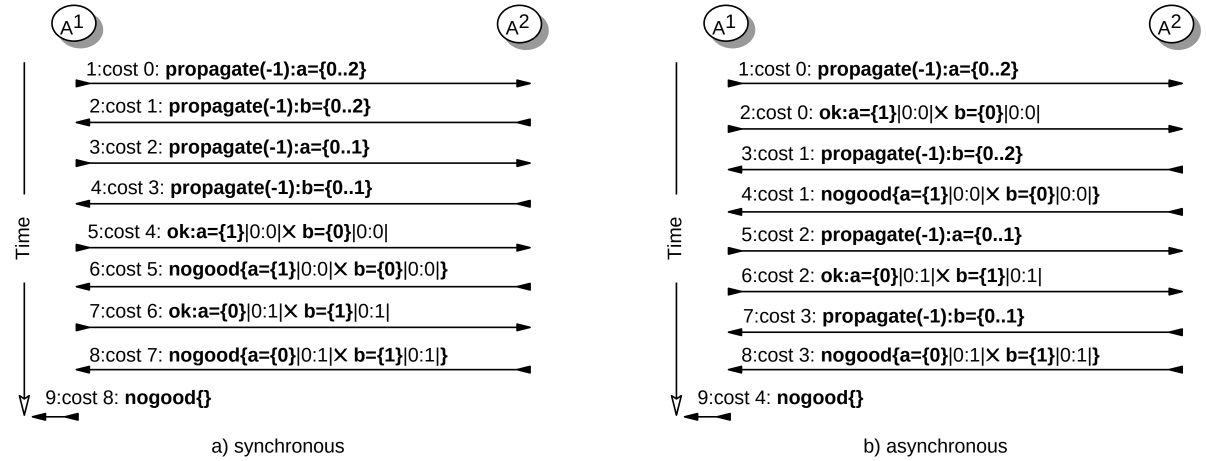

Figure 3: *Traces of message passing for the example in Fig 2.*

given branch of the search tree can also be used as soon as available. It is then desirable to allow a certain degree of asynchronism between search and bound-consistency maintenance so that the "best" behavior has chance to emerge first. If we consider for example the case of easy problems where no backtracking is required and where bound-consistency maintenance causes no domain wipe-out, running search and consistency asynchronously is clearly desirable since the former process no more has to wait for the useless convergence of the latter. Let us now illustrate our purpose on a basic situation where asynchronism helps. We consider a simple example with two agents handling a single constraint each. The constraints involve the same variables a and b and are shown in Figure 2.

Figure 3 compares the traces of message passing for this example between the synchronous and asynchronous variants of maintaining distributed consistency. The basic algorithm behaves as AAS except that, in addition to the `ok()` and `nogood()` messages, it also sends `propagate` messages which inform the agents about domain reductions computed by the DHC processes. In Figure 3, **propagate(k):x=**$\{$**a..b**$\}$ is a message informing the receiver that the domain reduction $[a, b]$ has been computed by a process DHC of level k for the variable x. The cost refers to the number of sequential messages required. In *synchronous MDC* four sequential messages are exchanged before the convergence of DHC. Still, the same amount of search remains to be done. Figure 3b shows that the whole search space can be exhausted with the same number of messages by AAS performed in parallel with DHC in asynchronous MHDC. In both cases, the messages for termination detection are not taken into account. The *synchronous MDC* requires more messages for termination detection since it requires detecting the termination of distributed consistencies.

5. THE MHDC ALGORITHM

In this section we will detail the MHDC algorithm. We start by giving the necessary background and definitions. As in AAS, agents in MHDC are assigned priorities. We assume that the agent A^i has priority over another agent A^j if $i > j$. A *link* exists between two agents if they share a variable. The link is directed from the agent with lower priority to the agent with higher priority. Let A^i and A^j be two agents related by a link such that $i < j$. A^i is called the *predecessor* of A^j and conversely, A^j is called the *successor* of A^i. The *system agent* (or broker) is a special agent that receives the subscriptions of the agents for the search. It decides the order of the agents, initializes the links and announces the termination of the search.

The notions of *assignment*, *aggregate* and *explicit nogoods* are defined as in AAS:

Definition 1 (Assignment) *An assignment is a triplet* (x_j, set_j, h_j) *where* x_j *is a variable,* set_j *a set of values for* x_j *and* h_j *a history of the pair* (x_j, set_j).

When MHDC incorporates bound-consistency, set_j is a range of ordered values. The history provides the information necessary for a correct message ordering. It determines if a given assignment is more recent than another. Let $a_1 = (x_j, set_j, h_j)$ and $a_2 = (x_j, set'_j, h'_j)$ be two labels for the variable x_j. a_1 is *newer* than a_2 if h_j is more recent than h'_j (see [10] for more details).

Definition 2 *An* aggregate *is a list of assignments.*

Definition 3 (Explicit nogoods) *An explicit nogood has the form* $\neg A$*, where* A *is an aggregate.*

5.1. THE MESSAGES

The agents exchange information about variables, assignments and conflicts with constraints (nogoods) using `ok()`, `nogood()`, `addlink` and `update()` messages:

- `ok()` messages have as parameter an aggregate. They represent proposals of domains for a given set of variables and are sent from agents with lower priorities to agents with higher priorities.
- `nogood()` messages have as parameter an explicit nogood and are sent from agents with higher priorities to agents with lower priorities.
- `addlink`(vars) messages: are sent from agent A^j to agent A^i (with $j > i$). They inform A^i that A^j is interested in the variables $vars$.
- `update()` messages: are sent as answers to `addlink` messages and for proposing domain changes for variables required with `addlink` messages. Their argument is an aggregate.

In addition, the agents also exchange information about the nogoods (domain reductions) inferred by bound-consistency maintenance. This is done using `propagate` messages which take lists of labels as arguments.

Definition 4 (Label) *A label generated by the agent* A^i *at consistency level* k *for the variable* x*, and denoted by* $\mathtt{label}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$*, is a triplet (*$x$*,* $\mathtt{r}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$*,* $\mathtt{context}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$*).* $\mathtt{r}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$ *is a range of values computed by a process* $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{k}}$ *for the variable* x*.* $\mathtt{context}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$ *is the context in which* $\mathtt{r}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{x})$ *is generated.*

The context is an aggregate which contains all the variables implied in the computation of $\mathtt{r}_\mathtt{k}^\mathtt{i}(\mathtt{x})$ by $\mathtt{DHC}_\mathtt{k}^\mathtt{i}$. It is used for updating the local information of the receiving agent, checking the validity of the propagated nogood and inferring nogoods after search or a consistency maintenance process has detected domain wipe out. It will be described in more details later.

Definition 5 (Labeling) *A labeling generated by the agent A^i at consistency level k, and denoted $\mathtt{labeling}_\mathtt{k}^\mathtt{i}$, is a list of labels $\mathtt{label}_\mathtt{k}^\mathtt{i}(\mathtt{x})$.*

We denote by $\mathtt{propagate}_\mathtt{k}^\mathtt{i}()$ the message sent by the agent A^i at level k. Such a message has the following characteristics:

- A $\mathtt{propagate}_\mathtt{k}^\mathtt{i}()$ message can only be sent by an agent A^i with $i > k$.
- A $\mathtt{propagate}_\mathtt{k}^\mathtt{i}()$ message takes a labeling as parameter. It is sent to agents A^j with $j \geq i$. A $\mathtt{propagate}_\mathtt{k}^\mathtt{i}()$ message informs the concerned agents about nogoods (domains reductions) inferred by a $\mathtt{DHC}_\mathtt{k}^\mathtt{i}$ process. The agent A^i can run a $\mathtt{DHC}_\mathtt{k}^\mathtt{i}$ process, $k \geq 0$, as soon as it has received an `ok()` message from the agent A^k or a $\mathtt{propagate}_\mathtt{k}^\mathtt{j}()$ message from any agent A^j with $j > k$.
- The labels, $\{(x_k, r_k, c_k) | k \in V^i \cap V^j\}$, sent by A^j to A^i only contains variables x_k whose r_k has just been modified by A^j and that are local variables common to A^i and A^j.

Each agent A^i maintains a stack S^i of labelings for its local variables. The stack's entry $S^i(k)$ of level k, with k varying from -1 to $i-1$, contains the labeling computed by the process $\mathtt{DHC}_\mathtt{k}^\mathtt{i}$. By construction, the domain allowed by $S^i(k)$ for a variable v is contained in the domain for v in $S^i(k-1)$.

5.2. THE LOCAL SEARCH SPACE

The *current solution space* of A^i, denoted as C_{A^i}, is described by the local constraints, the last entry of the stack S^i, a list of explicit nogoods, and a *view*.

As in AAS0, the explicit nogoods in MHDC can be merged into a single nogood using the following relaxation rule:

$$\begin{array}{c} V_1 \wedge V_2 \rightarrow \neg T^1 \\ V_1 \wedge V_3 \rightarrow \neg T^2 \\ \hline \Rightarrow \quad V_1 \wedge V_2 \wedge V_3 \rightarrow \neg(T^1 \vee T^2), \end{array} \qquad (1)$$

where V_1, V_2 and V_3 are aggregates, obtained by grouping the elements of the nogoods, such that they have no variable in common. The merged list of explicit nogoods then reduces to a a single nogood where the right part corresponds to the expanded tuples and the left one to a *conflict list* (CL).

The variables A^i is *interested in*, are its local variables and those establishing links with other agents.

Definition 6 (View) *The view of an agent A^i is an aggregate (V, R, H) such that V contains variables A^i is interested in.*

A view imposes restrictions on the original search space defined by the local constraints of an agent. It contains for each variable, the domain in the newest received assignment via `ok()` messages.

Definition 7 (Nogood entailed by view) *Let V_1 be the view of a given agent, T be the set of tuples disabled from the original solution space by V_1. We say that the nogood $V_1 \rightarrow \neg T$ is entailed by the view V_1.*

Definition 8 (Nogood entailed by consistency) *Let T be the set of tuples disabled from the original solution space by the labeling L of $S^i(k)$. We say that the nogood $L \rightarrow \neg T$ is entailed by consistency.*

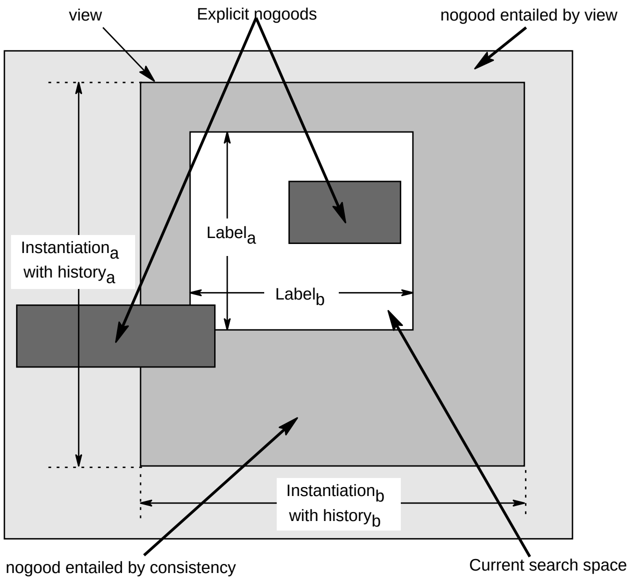

Figure 4: *The local information maintained by each agent A^i. $Instantiation_a$ and $Instantiation_b$ corresponds to the latest instantiations of variables a and b and define the view. $Label_a$ and $Label_b$ are the latest computed bound consistent ranges for a and b on the top of S^i. Any new instantiation must be generated in the white box and satisfy the local constraints of the agent.*

The set $\{\mathtt{labeling}_\mathtt{k}^\mathtt{i} \rightarrow \neg T\}$ with $\mathtt{labeling}_\mathtt{k}^\mathtt{i} \in S^i(k)$ and $k = \{-1 \ldots i-1\}$ contains all the nogoods entailed by consistency for a given agent A^i.

A tuple is *contained* in the current solution space of agent A^i if it satisfies the local constraints and is not contained in the explicit or entailed nogoods of C_{A^i}. The *current instantiation* of an agent A^i is an assignment such that all its tuples are contained in C_{A^i}. Figure 4 schematizes the local information maintained by each agent on a two dimensional example for the top of the stack.

The list of nogoods, respectively the view, of an agent A^i is updated by the `nogood()`, respectively `ok()` messages it receives. The stack of labelings is modified using `propagate` messages or `ok()` messages. More precisely, the stack of labeling is updated using the nogoods specified by the labels. These nogoods are given by the difference between the context information and the range in each label. The context, $\mathtt{context}_\mathtt{k}^\mathtt{i}(\mathtt{x})$, of a label (x,$\mathtt{r}_\mathtt{k}^\mathtt{i}(\mathtt{x})$, $\mathtt{context}_\mathtt{k}^\mathtt{i}(\mathtt{x})$) is simply the subset of the view of A^i used for computing $\mathtt{r}_\mathtt{k}^\mathtt{i}(\mathtt{x})$ using a $\mathtt{DHC}_\mathtt{k}^\mathtt{i}$ process.

Definition 9 (Obsolete assignment) *An assignment (x_j, r_j, h_j) received by an agent A^i is obsolete if the view of A^i contains a newer assignment for x_j.*

Definition 10 (Valid label) *A label sent in a* `propagate()` *message is* valid *if its context contains no obsolete assignment.*

5.3. THE BACKTRACK PROCEDURE

In MHDC, the core backtrack procedure for each agent is given by the finite state machine of Figure 5. Each agent A^i starts by initializing its local information, running a local bound-consistency algorithm and launching a DHC process of level -1. Then, it will essentially transit between two main states, the state `PROCESSING` where it tries to build a local solution and `ANNOUNCED` where it has either succeeded or failed in building a local solution and is waiting for new messages.

From both states, several transitions can be selected depending on the type of message received. The transitions labeled by `propagate(k)`, `ok`, `update` and `nogood` are selected upon reception of `propagate`, `ok()`, `update()` and `nogood()` messages respectively. Transitions are assigned priorities in such

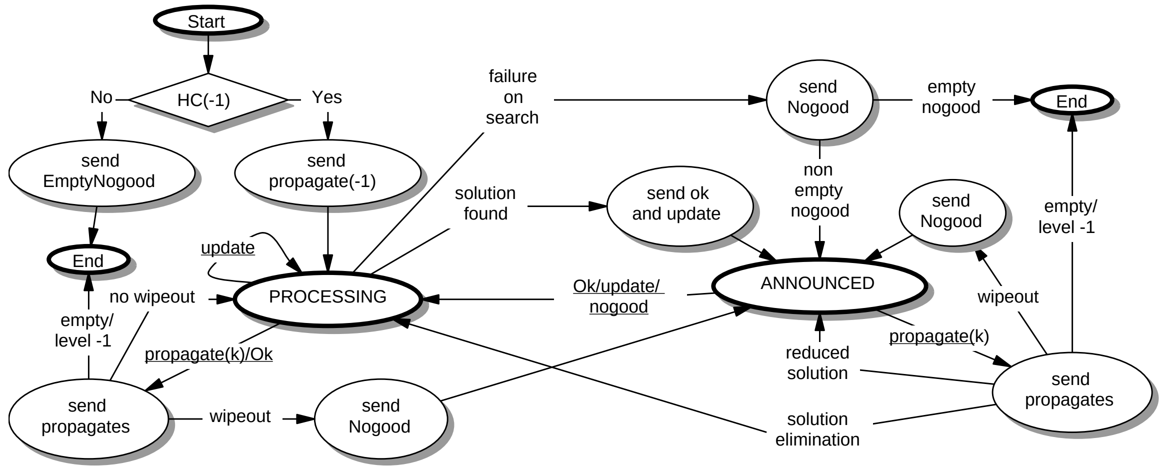

Figure 5: *Backtrack search procedure for each agent.*

a way that `ok` and `update` from agent A^k or `propagate(k)` with the lowest k are selected first, then `nogood` transitions.

When in the state `PROCESSING` an agent A^i receives no message, it tries to generate a current instantiation from C_{A^i}. We say that it is *searching*. If a local solution is found (i.e. a set of tuples can be extracted from C_{A^i}), the agent announces the instantiation by sending `ok()` and `update()` messages to the concerned agents and transits into the state `ANNOUNCED`. When at the contrary C_{A^i} becomes empty during search, A^i announces a nogood and also transits into the state `ANNOUNCED`. At any time, the search process can be interrupted by the reception of an `ok()`, `propagate` or `nogood()` message[5]. This cause the agent to execute the procedure Update-search-space in the two former cases and Nogood in the latter. These procedures update the local search space (i.e. the views, the nogoods lists, the stack of labels and the position in the search tree) according to the content of the message. The current instantiation of an agent A^i is known as long as it remains in the state `ANNOUNCED`.

The state `send propagates` is reached after a `propagate(k)` transition has been selected and the procedure Update-search-space executed, which possibly results in a new labeling L for the local variables of A^i. If L contains an empty label l (domain wipe-out), a `nogood(`$\neg context(l)$`)` message is generated and A^i transits into the state `ANNOUNCED`. Otherwise, A^i sends $\mathtt{propagate}^{\mathtt{i}}_{\mathtt{j}}()$ messages, with $j \geq k$, to all the concerned agents. If A^i has a current instantiation and if this instantiation is eliminated by L, A^i comes back to the state `PROCESSING`. If the current instantiation is modified, the agent may send `ok()` and `update()` messages according to the rules of AAS.

The procedure Nogood (Figure 8) treats incoming `nogood()` messages. The argument, Q, of such a message is an explicit nogood. Let V be the view of the receiving agent. Suppose that there exists in Q, respectively in V, an assignment a_1, respectively a_2 for the variable x_j such that a_1 is newer than a_2. We will say that the nogood gives a *new view* for the variable x_j. In this case, the agent has to update its view by sending an `update()` or `ok()` message to itself. An explicit nogood is valid if it concerns (i.e. invalidates) the current instantiation of the agent. If the received nogood is valid and if it contains variables that are unknown in the current view of the agent, `addlink` messages will be sent to establish new links with all the predecessors for which these variables are local.

[5] In the particular case of AAS0, since the nogoods are not stored, the `nogood()` messages are treated only in the state `ANNOUNCED`, when the agent has a current instantiation, and discarded otherwise.

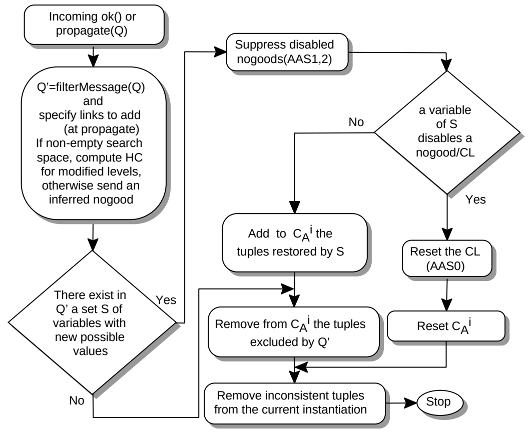

Figure 6: *Update-search-space procedure.*

The procedure Update-search-space (Figure 6) has a labeling Q as argument. It treats incoming $\mathtt{propagate}^{\mathtt{i}}_{\mathtt{k}}(\mathtt{L}^{\mathtt{i}}_{\mathtt{k}})$ or `ok(A)` messages. In the first case, Q corresponds to L^i_k, otherwise Q is constructed by replacing each assignment $(v_j, r_j, hist_j)$ of `A` by a label $(v_j, r_j, ((v_j, r_j, hist_j)))$. The procedure Update-search-space starts by checking whether there exist in the contexts of the labels, assignments, (x_j, r_j, h_j) newer than their counterparts in the current view. If this is the case, the labels of the stack depending on histories anterior to h_j are disabled. The labels disabled at the level j of S^i are iteratively reset using the information of level $j-1$. Then, the invalid labels are filtered out of Q and the valid ones are combined with their counterparts in the stack. This combination can lead to lost of valid nogoods, which triggers in its turn one of the conditions for sending `ok()` messages as mentioned in [10]. The procedure Update-search-space then launches a $\mathtt{DHC}^{\mathtt{i}}_{\mathtt{k}}$ process using the labeling of $S^i(k)$ and proceeds to updating the set C_{A^i}, as well as the current instantiation of the agent A^i according to the new labeling of $S^i(k)$. Suppose that one of the labels of $S^i(k)$ offers a new possibility of valuation for a non-local variable x_j with respect to the current view. In AAS2 or AAS1 all the nogoods which do not take the new possibility into account will be *disabled*. In AAS1 this means that they will be removed. In AAS2 they will be marked and kept for an eventual further usage. In AAS0, if the nogood obtained by the relaxed inference rule contains such a variable but does not take the new value into account, the conflict list will be reset. Resetting C_{A^i} means that all the tuples allowed by the current nogoods and view are introduced in C_{A^i}. In the end, the previous instantiation can be updated and renewed.

The procedure Update (Figure 7) is similar to

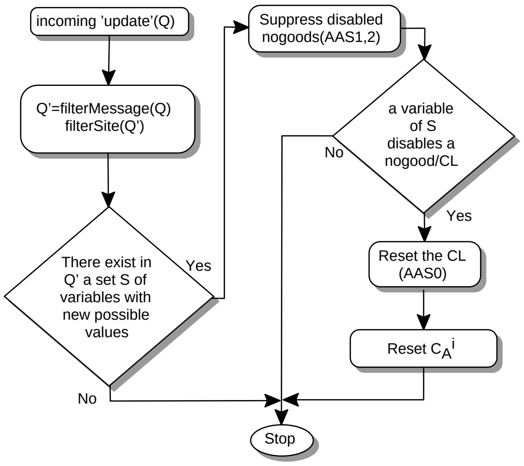

Figure 7: *Update procedure.*

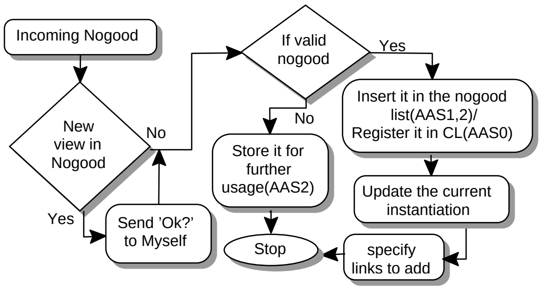

Figure 8: *Nogood procedure.*

Update-search-space. It treats incoming `update()` messages. An `update(A)` message is equivalent to an `ok(A)` message where `A` contains assignment for variables that are non-local to the receiving agent A^i. Since the assignment concerns external variables, the current instantiation does not need to be updated. This constitutes the only difference between Update and Update-search-space.

In order to ensure a short time for the preemption of lower priority search on the arrival of higher priority messages, an interruptible backtracking procedure is used (see the annexes of section 11).

5.4. FLOW CONTROL

Several incoming messages can contribute to the same level of consistency. Treating them individually, combined with the fact that each agent strives to do as much as it can, would result in uselessly fragmenting the local consistency output. Furthermore, since the time needed to send and process a messages is non zero agent buffers would risk to run short of memory. For this reason, our agent architecture (see 1) offers the opportunity to temper the flood of messages by compacting incoming messages. By this means, several messages of given types can be answered in block. This way, not only that the total number of messages is reduced, but also the space in local incoming buffers is spared, since all the messages are compacted in the receivers. For `propagate`, `update()` and `ok()` messages there is a common compactor for each k. The `nogood()` messages are received by the compactor with the same ID as the one of the current agent.[6]

The proposed "flow control" policy uses a threshold t. When the total size of the outgoing queues increases over t, the output of the compactors towards the scheduler are interrupted, reducing this way the total number of messages in the network. In the same idea, if the total size of the incoming queues grows over a certain threshold t^1, the output of the compactors can be blocked. This gives more priority to the compacting process until the size decreases to a lower threshold. The server can refuse connections as long as the total size of the incoming queues grows over a certain threshold $t^2 > t^1$. Connections are refused during a limited time since the size of the input queues necessarily decrease due to blocking.

Since overwhelming "avalanche" of messages appears seldom and temporarily in search processes, the proposed "flow control" is needed and induces no significant overhead. As presented, the compactors give priority to lower levels of important information.

[6] There is also one compactor for `accepted` messages (see [10]).

6. THEORETICAL EVALUATION

MHDC builds on AAS which is proven to be correct and complete and terminates [10]. MHDC is AAS with the inference and transmission of the additional nogoods generated by bound-consistency maintenance. As argued in [10] for AAS1 and AAS2, the use of additional nogoods in the local decisions maintains the correctness, termination and completeness properties.

To insure that the strength of the consistency maintained in MHDC and *synchronous MDC* are strictly equivalent, agents using MHDC need to maintain all the valid nogoods entailed by consistency that they have received.

Under these conditions, if the threads with higher priority are treated completely before the lower priority ones using an ideal local priority-based scheduler, then MHDC cannot perform worse in time than *synchronous MDC* with ideal termination detection. The proof is straightforward. The difference between MHDC and *synchronous MDC*, provided a perfect priority-based scheduler, is that the former will send, receive and analyze some future messages in the idle moments of the agents. Therefore, even if in some cases, those future messages would have never been generated by *synchronous MDC* due to `DHC` domain wipe-outs, they cannot reduce the performance with an ideal scheduler. They are treated only in idle time and are eliminated as soon as they become out of date.

However, our scheduler is not perfect since it needs time to decide priorities and to preempt running threads. In practice, the worst running time of MHDC can therefore be higher than the one of *synchronous MDC* with a fraction equal to the ratio between the longest time needed in order to preempt a thread and the shortest time needed to perform a higher priority task. However this ratio is upper bounded for any MHDC implementation. Moreover, the termination detection of DHC in the synchronous approach cannot be done instantaneously and may require many messages.

The algorithm was presented by means of the `ok()`, `nogood()`, and `update()` messages in order to present a general framework where different consistency algorithms can be used and where the previous algorithms are easily recognized. However, the messages `propagate` and `addlink` are enough general to sufficiently model the MHDC algorithm.

Our current implementation of MHDC is a simplified version, where the valid nogoods entailed by consistency are merged (see section 5.3). The maintained consistency can therefore be weaker than in *synchronous MDC*. Further work is planned to compare *synchronous MDC* with different variants of MHDC.

7. EXPERIMENTS

We have analyzed MHDC built on AAS0. We have generated a set of random distributed problems with a quantifiable complexity. Since the complexity of random centralized problems is well studied, we have chosen to generate our distributed problems starting from centralized problems with known complexities.

More precisely, we generate random centralized problems (CP) with n variables and having their density and tightness chosen in the peak of complexity. We then circularly distribute the constraints to each agent, one constraint at a time.

In order to chose the constraint to be attributed to an agent we

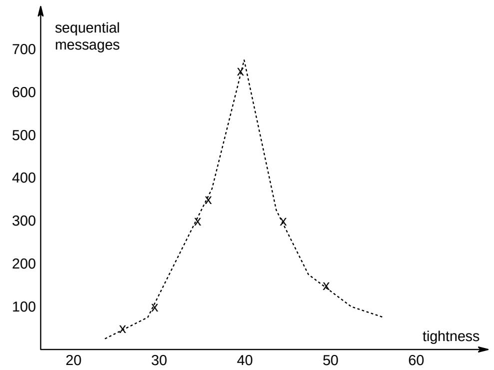

Figure 9: *Results averaged on more than 100 examples per test point with 25 agents and 20 variables.*

pick randomly a constraint from the remaining ones of CP. We allow a number $\tau_1 n$ of trials to pick one constraint that had both its variables in the current agent. On failure we allow a number $\tau_2 n$ trials for picking one constraint that had at least one of its variables in the current agent. If this second chance has failed, then we simply pick a constraint at random.

Once a constraint has been chosen for the current agent, we add it to the local CSP of that agent and remove it from CP. Thus we have two new parameters for the complexity of the obtained distributed problems, namely τ_1 and τ_2. These parameters quantify the effort for clustering the variables within agents. High values for τ_1 and τ_2 lead to groups of constraints among a limited number of variables, (choosing cliques in each agent). Instead, low values for τ_1 and τ_2 lead to agents interested in most variables. In our experiments we used $2\tau_1 = \tau_2 = 1$. The preliminary results have been obtained with 25 agents situated on distinct computers on a LAN. They were solving CSPs with 20 variables of 8 elements, at a density of 20% and variable tightness. The results are given in the figure 9. We have performed more than 100 tests for each of 7 chosen densities around the peak. We mention that on these kind of problems, AAS was too slow to allow comparison. In [10], AAS has been tested on problems with 20 agents, 20 variables and 5 values per domain. Increasing the size of the domains or the number of agents was strongly increasing the number of sequential messages.

8. DISCUSSION

8.1. CONSTRAINT RELAXATION FOR DCSPS WITH PRIVATE CONSTRAINTS

DCSPs with private constraints fit well problems implied in negotiations. However, it may happen that these problems are over-constrained or that the agents have different preferences for some solutions. Max-CSPs and Valued-CSPs [14] have already been used to tackle this problem in the context of public constraints and honest agents. We now briefly discuss how constraint relaxation can be incorporated into MHDC. Since the constraints of each agent are private, an agent can be tempted to create fake or modified constraints in order to mislead the others. In order to give equitable chances to each agent in the case of over-constrained DCSPs, we must then consider that each agent owns exactly one global constraint.

The relaxation can be achieved following a game-theory schema. Whenever the search fails to find a solution, a deadline is given to the agents for the submission of a message containing tuples that the sending agent has rejected as nogood in the previous search and it would accept in a new one. The messages have to be opened simultaneously. If no agent relaxes its constraints, the search fails. If some agents have submitted nonempty messages, the agents are reordered by giving higher priority to the yielding agents. Search is then performed by cutting the branches containing no new tuple.

The previous strategy can be followed until failure or a solution is found. Some problems are posed by this approach. The first one is related to the submission of messages with new tuples. A cryptographic technique may be a solution. The second problem consists in the detection of lying agents, namely agents that submit tuples that were not included in their local search space or not in conflict with their local CSP in the previous search. A trace of all the generated nogoods may be required and a space problem can appear for storage in hard problems. It may be relaxed by checking that the previous agents could have accepted that combination.

Another open problem is the reuse of stored nogoods. To help other agents reuse nogoods, an agent can store in the context of the nogoods it generates (either explicit nogoods or nogoods entailed by consistency) the reference (CR) of any local constraints used for the inference. When the agent renounces at some local constraints, it will offer a method to tests whether the CRs in the stored nogoods of other agents have expired. If the agents tend to maintain secret the number of the constraints to which they could renounce, they will use new CRs each time they refer to any internal constraint, even if it is a constraint on which they cannot yield. The eventual empty nogoods generated this way shows whether a given relaxation has chances to solve the problem.

However, the space complexity is being modified. If the agents are allowed to generate a high number of CRs then each agent has to be allowed to substitute subsets of the list of CRs in a nogood by a generic CR covering the global problem or by an CR covering the local problem of one agent.

8.2. TERMINATION DETECTION

The method used for termination detection in AAS (see [10]) has been improved by sending `accepted` messages only on arcs forming a spanning tree of the directed graph on which they were sent in [10]. We choose only one arc along which an agent sends `accepted` messages. This arc leads to the nearest priority agent among those to which `accepted` messages were sent in [10].

8.3. AGENT REORDERING

Agent reordering, as the one presented in [15], is an important issue. We conjecture that AAS can easily be extended to the reordering algorithms presented in previous work [15]. Such extensions can bring interesting improvements, specially for the incomplete versions of the AWC algorithms where not all the nogoods are stored. The extension would consist of allowing all the agents to modify all the variables for which they have constraints and allowing them to propose spaces of solutions instead of single values.

However, the integration of agent reordering schemes in MHDC has do be done with more care. It requires the switch of context in the computation of global consistency.

9. CONCLUSIONS

We have presented MHDC, a new distributed search technique which allows for maintaining distributed consistency with a high degree of parallelism and without resorting to intermediate termination detection. MHDC is based on AAS and thus provides a natural support for enforcing privacy requirements on constraints [11]. The preliminary evaluation has been done with a version based on AAS0 which, consequently, maintains a minimal number of nogoods. The experiments have shown that the overall performance of MHDC is significantly improved compared to that of AAS [10]. MHDC has much potential in practice. It accommodates a higher number of agents than AAS, requires a bounded local space, reduces the number of messages needed for termination detection, and improves parallelization compared to *synchronous*

MDC. MHDC fully exploits the aggregation capability of AAS. If built on AAS0, MHDC guarantees polynomial space complexity.

10. ACKNOWLEDGMENTS

Most of the MELY search agent platform was implemented with the help of Michel Galley. We also want to thank the reviewers for their comments. This work was performed at the Artificial Intelligence Laboratory of the Swiss Federal Institute of Technology in Lausanne and was sponsored by the Swiss National Science Foundation under project number 21-52462.97.

REFERENCES

[1] J.G. D'Ambrosio, T. Darr, and W.P. Birmingham. Hierarchical concurrent engineering in a multiagent framework. *Concurrent Engineering: Research and Applications (CERA) - An International Journal*, 4(1):47–57, March 96.

[2] E.H.Turner and J.Phelps. Determining the usefulness of information from its use during problem solving. In *Proceedings of AA2000*, pages 207–208, 2000.

[3] Y. Hamadi. Optimal distributed arc-consistency. In *Proceedings of CP'99*, Oct 99.

[4] Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 98.

[5] K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of ICMAS-2000*, 00.

[6] J.Silva and T.Glass. Combinational equivalence checking using satisability and recursive learning. In *Proceedings of the ACM/IEEE Design, Automation and Test in Europe Conference*, pages 145–149, 1999.

[7] A.K. Mackworth. Consistency in networks of relations. *Artificial Intellegence*, 8(1):99–118, 77.

[8] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 94.

[9] M.-C. Silaghi. Synchronous Java with Priorities. http://liawww.epfl.ch/˜silaghi/sJavap, 2000.

[10] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, Austin, August 2000.

[11] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.

[12] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Fractionnement intelligent de domaine pour CSPs avec domaines ordonnés. In *Proc. of RFIA2000*, 2000.

[13] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Ways of maintaining arc consistency in search using Cartesian representation. In K.R. Apt, A.C. Kakas, E. Monfroy, and F. Rossi, editors, *"New Trends in Constraints" (Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999)*, number 1865 in LNAI. Springer, 2000.

[14] M. Yokoo. Constraint relaxation in distributed constraint satisfaction problem. In *ICDCS'93*, pages 56–63, June 93.

[15] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, September/October 1998.

[16] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS'92*, pages 614–621, June 92.

[17] Z.Collin, R.Dechter, and S.Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI 1991*, pages 318–324, 91.

[18] Y. Zhang and A.K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 91.

11. ANNEXES

In this annexes we give some procedures used in local computations. The procedure *filterMessage* filters the incoming information via propagate messages, given the local knowledge gathered in the corresponding compactor (superscripts DHC(k)). $\mathtt{hist}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})$ is used to note the history of the assignment of level k for the variable v that is known by the agent A^i. *request-filter-site* prepares the structures for filtering the structures on site.

Procedure `filterMessage`($\mathtt{labeling}_{\mathtt{k}}^{\mathtt{i}}$) : Filter out outdated information.

foreach *(*$\mathtt{label}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})\in\mathtt{labeling}_{\mathtt{k}}^{\mathtt{i}}$*)* **do**
  foreach *(*$\mathtt{hist}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{var})\in\mathtt{context}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})$*)* **do**
    if *(*$\mathtt{hist}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{var})$ *newer than* $\mathtt{hist}_{\mathtt{k}}^{\mathtt{DHC(k)}}(\mathtt{var})$*)* **then**
      request-filter-site(changed,$\mathtt{hist}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{var})$);
  foreach *(var*$\in\mathtt{context}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})$*)* **do**
    if *(*$\mathtt{hist}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{var})$ *older than* $\mathtt{hist}_{\mathtt{k}}^{\mathtt{DHC(k)}}(\mathtt{var})$*)* **then**
      remove $\mathtt{label}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})$ from $\mathtt{labeling}_{\mathtt{k}}^{\mathtt{i}}$;
      break;
  remove changed from DHC(k);
  integrate $\mathtt{label}_{\mathtt{k}}^{\mathtt{i}}(\mathtt{v})$ in the compactor $DHC(k)$;

The procedure *Interruptible Backtracking* presented now is used on dual local CSPs[7] and is obtained by restructuring the usual recursive backtracking procedure. *crtConstraint* stands for the index of the current constraint and is initially 0. *UpdateDomains* is the procedure that sets the current domains to the domains allowed after analyzing the constraint $crtConstraint - 1$. *nextInstantiation* is a procedure that looks for a next valid set of tuples in *crtConstraint* and returns true on success after updating the allowed domains. The *resetConstraint* procedure resets the iterator called by *nextInstantiation*. *checkFinished* checks whether we have checked all the constraints. An interrupt sets the *end* flag.

[7]The nodes are constraints.

Procedure `search`: Interruptible Backtracking.

```
result=INTERRUPTED;
while (!end) do
    UpdateDomains(crtConstraint);
    if (!nextInstantiation(crtConstraint)) then
        if (crtConstraint==0) then
            crtInstantiated = false;
            result=EXAUSTED;
            resetConstraint(0);
            end=true;
        else
            crtConstraint−−;
        continue;
    crtInstantiated=true;
    if (!modif||mHC(crtConstraint)) then
        crtConstraint++;
        crtInstantiated=false;
        if (checkFinished(crtConstraint)) then
            crtConstraint−−;
            crtInstantiated=true;
            result=SOLUTION;
            end=true;
        else
            resetConstraint(crtConstraint);
    else
        crtInstantiated=false;
return result;
```