

# Descending Requirements Search for DisCSPs

Michael Orlov<sup>1</sup> and Amnon Meisels<sup>2</sup>

**Abstract.** A new search algorithm, Descending Requirements Search (DESRs), for Distributed CSPs is proposed.

The algorithm is composed of two independent phases. In the first phase, agents form a binary hierarchy of groups. The distributed partition algorithm uses a heuristic that prefers to join neighbors that are strongly constrained, into groups. This is done concurrently at all levels of the hierarchy.

In the second phase, concurrent independent backtracking search processes grow partial assignments along a hierarchy of agent groups, each agent participating in multiple search processes. The order of assignments is partially determined by the hierarchy of groups. Independent partial solutions are grown by agents, each partial solution is sent higher up in the hierarchy, ultimately resulting in the top-level agent producing a solution, or in some agent producing an empty *Nogood*.

The new algorithm is evaluated experimentally on randomly generated DISCSPs. Both run-time performance and network load of DESRS are better than Asynchronous Backtracking (ABT). The run-time performance of DESRS is similar to that of the best concurrent search algorithm CONCDB, on medium sized problems.

## 1 Introduction

Distributed search algorithms for distributed constraint satisfaction use the concurrency of multi-agent execution in many forms. One popular way of concurrency is to let all agents participate in a single backtracking search, by operating asynchronously. In asynchronous backtracking (ABT) [2, 10], agents assign their variables asynchronously and check for consistency by sending forward *ok?* messages. Backtracking operations are performed by sending *Nogoods*, and for the correctness of the algorithm a fixed order of agents is essential [10].

Another form of achieving concurrency for DISCSP search is to use multiple search processes. Concurrent dynamic backtracking (CONCDB) utilizes multiple concurrent search processes on a distributed CSP [11–13]. CONCDB maintains a dynamic number of backtracking search processes and generates an efficient concurrent performance [13].

The present paper proposes to use cooperation among multiple concurrent search processes, each searching a *partial search space*. The proposed algorithm uses a hierarchy of groups of agents that search concurrently for partial solutions. In *Descending Requirements Search* (DESRs) groups of agents communicate in order to maintain consistency and arrive at a consistent solution. Each group in the constructed hierarchy is represented by one of its members.

The representative agents (i.e., *leaders*) compute the consistent partial solutions for the agents that form the group. Naturally, an important part of this hierarchical search algorithm is the smart partition of the agents in the hierarchy of groups.

DESRs is composed of two phases. In the first phase, agents generate a hierarchy of groups and select representatives for each group. Representative agents maintain partial solutions of the group they represent and connections with other groups. In the second phase of the algorithm, the hierarchy of groups of agents searches concurrently for multiple solutions of the DISCSP.

The grouping of agents generates partial orders of assignments among agents. The hierarchy takes the form of a binary tree. Solutions are generated by passing compatible partial solutions among all agents. Consequently, the grouping results in a partial order among agents which incrementally generate the solutions.

DESRs generates only compatible partial solutions. Agents *extend* partial solutions generated by other agents or groups. The leaders of groups at all levels route partial solutions of one of their components to their other components, to be extended in a consistent way. Requirements for extending partial solutions are being *routed down* (i.e., descended) by the leaders of groups.

Descending requirements search does not impose a total order on the agents or variables of the problem, just the partial order that is implicit in the partition into groups. It is complete and correct, and provides multiple solutions to the DISCSP. DESRS maintains multiple concurrent search procedures, that are coordinated within each group (level) in the hierarchy by the leader of that group. By maintaining multiple search processes concurrently, search can be made more efficient, especially if multiple solutions are needed.

The algorithm is evaluated experimentally on randomly generated DISCSPs [5, 6, 8], and compared to existing search algorithms — ABT [10] and CONCDB [13]. The DESRS algorithm is shown experimentally to perform better than ABT on all problems. The concurrent runtime performance of DESRS is similar to that of CONCDB on problems of limited complexity.

Section 2 describes the first phase of DESRS — partitioning of agents into a hierarchy of groups. Section 3 describes the solving phase of the algorithm. In Section 4, an experimental evaluation of DESRS on randomly generated problems is presented. Section 5 presents our conclusions, including the discussion of the distributed partitioning algorithm, which is of interest by itself.

## 2 Group partitioning

The idea at the basis of hierarchical search is to prune inconsistent partial assignments by concurrent processes of computation. That is, consistent partial assignments are produced concurrently by groups of agents, and then united into larger consistent assignments. Agents are divided into a hierarchy, where each agent belongs to groups at

<sup>1</sup> Supported by the Lynne and William Frankel Center for Computer Sciences and the Paul Ivanier Center for Robotics Research and Production Management.

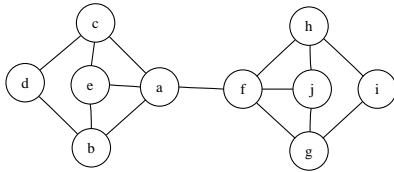
<sup>2</sup> Ben-Gurion University, Israel, email: {orlov, am}@cs.bgu.ac.il.

different levels. Each group has a level, where groups at level  $i$  are composed of zero or two groups of level  $j < i$ . The result is a binary tree, where all non-leaf nodes have two children. There is exactly one group containing all agents (i.e., the entire DISCSP).

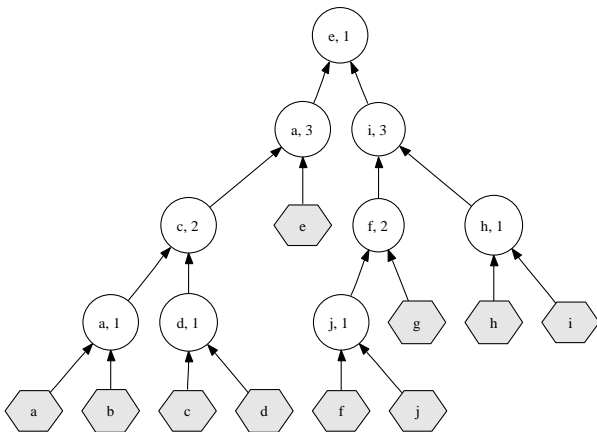
Each group has an agent that stores and manipulates the consistent partial assignments of the group. This agent is termed the representative agent, or the *leader*, of the group. Messages containing consistent partial assignments of a group are sent by the group's representative to the leader of the next higher level. In order to construct consistent partial solutions, each leader communicates with its group, in order to request generation of consistent partial solutions (Section 3).

The descending requirements search algorithm first partitions the DISCSP into groups of agents that form a hierarchy. This is done by merging pairs of agents or pairs of groups into higher level groups (Algorithm 1).

Consider the constraints network in Figure 1. Note that there is only one constraint between the two groups  $\{a, b, c, d, e\}$  and  $\{f, g, h, i, j\}$ . As a result, the merging of consistent partial assignments to these two particular groups will involve checking only this single constraint (connecting agents  $a$  and  $f$ ). A possible partition for this constraints network is given in Figure 2, which also specifies the number of constraints between neighboring groups. At level 0 of the tree in Figure 2 there are 6 agents. Three of the groups in level 1 include 2 agents each, and the other three are composed of a single agent —  $\{g\}$ ,  $\{h\}$ ,  $\{i\}$ . There are 4 groups at level 2:  $\{a, b, c, d\}$ ,  $\{e\}$ ,  $\{f, j, g\}$ ,  $\{h, i\}$ . There are 2 groups at level 3, and, the top-level group is at level 4.



**Figure 1.** Constraints graph with 10 nodes. Edges represent binary constraints.



**Figure 2.** Partitioning of the constraints graph of Figure 1. Non-leaf nodes show the representative agent which is responsible for the corresponding group, and the number of constraints connecting the merged groups.

The partitioning process starts with all agents at level 0. Pairs of agents are merged into level-1 groups, and pairs of level- $i$  groups are

merged into level  $i + 1$  groups. If an agent or a group is unable to merge with another group at some level, it becomes a group of one level higher. In this higher level, it is the only component. The root of the binary tree that is formed by the partition represents the complete DISCSP.

Note that at a first glance, it may seem that the partitioning shown in Figure 2 is actually “bad”, in the sense that pairs of agents or groups selected at the lower levels in the hierarchy are not the most constrained. This illusion stems from the general situation in which bigger groups have a greater total number of constraints between them. This is not true for the number of constraints *per agent*.

During the partitioning process each generated group designates one of its members to be a leader of the group. Since the partition follows the structure of a binary tree, each agent can be a representative of a single group. In other words, for  $n$  agents at most  $n - 1$  can be leaders. An example assignment of leaders is also shown in Figure 2. Agent  $c$  is the leader of a group with four components. It has two subgroups whose leaders are agents  $a$  and  $d$ . At the solving stage of the DESRS algorithm, agent  $i$  routes growing partial solutions produced by its two subgroups. The partial solutions are sent to agent  $i$  by agent  $f$  (which leads  $\{f, j, g\}$ ) and agent  $h$  (which leads  $\{h, i\}$ ).

Every agent performs two independent tasks. As a group representative it controls the production of consistent partial assignments of its group. As a local constraints network, it responds to requests for partial solution extensions. Ultimately, the root node agent produces globally consistent solutions. There are, in principle, two main ways to merge partial solutions. One way is for the leader to merge two partial solutions that were produced independently by its subgroups. Another way is for the leader to request its subgroups to *complete* partial solutions produced by the other subgroups. The latter approach is presented in the remainder of this paper.

Partitioning into groups is a distributed algorithm, such that upon its termination groups of agents are organized into a binary tree. All agents are leaves, and representative agents compose all higher levels of the binary tree (see Figure 2). Groups at the same level are disjoint, and groups of higher level contain groups of lower levels.

The main part of the algorithm is presented in Algorithm 1. The algorithm is run by each agent, and performs merges of pairs of agents or groups into ever larger groups. It runs concurrently by each agent selecting a neighbor to merge with. Merging is performed when two agents agree to merge.

The algorithm works as follows. Each agent  $s$  selects one of its neighbors (say,  $g$ ) as a candidate for merging with. If the selection is mutual, the agents merge. If not, the selected agent  $g$  can either respond negatively (i.e., NoJoin) when its own selection was different, or inform  $s$  that it has already made a merging decision (i.e., Done( $g, \dots$ )). Any merging decision is sent to all neighbors of the merging agent with Done messages. Agents form a queue of all their neighbors, ordered by their priority of merging with these neighbors, and erase from the queue the neighbors that have already merged. Agents stop proposing when their queues of neighbors are empty. In such a case, the agent advances to the next level without joining another group.

Partitioning proceeds in *levels*, where during each level every agent joins a neighboring agent, unless this is not possible. At the end, a binary hierarchy of groups is established, with the root group containing all the nodes of the network. During each level every agent continuously tries to join one of its neighbors, preferring agents with lower *connection weight*. This way, dense connectivity (i.e., constraints tightness) remains in lower levels of the hierarchy, and

---

**Algorithm 1:** GROUP-PARTITION( $s, N$ ): Partitioning into groups.

---

**Input** : agent  $s$ , its neighbors  $N$   
**Output**: components  $C$ , pairs, leader, parent  $\leftarrow$  USER  
**Locals** :  $\tilde{N}, N_{ldr}, g, level \leftarrow p, start \leftarrow TRUE, next-level$

```

1 SEND( $s, Leader(N, \{(s, N, 0)\}, TRUE, 0)$ )
2 loop forever do
3   switch RECEIVE() do
4     case Join( $t$ )
5       if  $t = g$  then
6         SEND( $g, Components(C)$ )
7       else
8         SEND( $t, NoJoin$ )
9     case NoJoin
10       $g \leftarrow SELECT(\tilde{N})$ 
11      SEND( $g, Join(s)$ )
12     case Components( $C_g$ )
13      if  $s \neq g$  then
14         $\{leader, C\} \leftarrow SELECT-LEADER(C, C_g, next-level)$ 
15        if parent = USER then
16          parent  $\leftarrow$  leader
17      else
18        leader  $\leftarrow$   $s$ 
19        foreach  $t \in N \cup \{s\}$  do
20          SEND( $t, Done(s, leader)$ )
21     case Done( $t, leader_t$ )
22      if  $t \neq s$  then
23         $N_{ldr} \leftarrow UPDATE(N_{ldr}, N, t, leader_t)$ 
24         $\tilde{N} \leftarrow \tilde{N} \setminus \{t\}$ 
25        if  $\tilde{N} = NIL$  then
26          level  $\leftarrow$   $p$ 
27          SEND( $leader, Leader(N_{ldr} \setminus \{leader\}, C, s = g, next-level)$ )
28     case Leader( $N', C', single, level'$ )
29      if start then
30         $N \leftarrow N', C \leftarrow C', N_{ldr} \leftarrow NIL$ 
31      else
32        pairs  $\leftarrow \{(t, r), w) : (t, \hat{N}) \in C \wedge r \in C' \wedge (r, w) \in \hat{N}\}$ 
33         $N \leftarrow COMBINE(N, N'), C \leftarrow C \cup C'$ 
34      start  $\leftarrow \neg start \vee single$ 
35      if start then
36        if  $N = NIL$  then
37          leader  $\leftarrow$  USER
38          foreach  $t \in C$  do
39            SEND( $t, Search$ )
40        else
41          level  $\leftarrow level', next-level \leftarrow level + 1$ 
42           $\tilde{N} \leftarrow N \cup \{(s, 1.5)\}$ 
43           $g \leftarrow SELECT(\tilde{N})$ 
44          SEND( $g, Join(s)$ )

```

---

inconsistent partial solutions are expected to be pruned as early as possible.

Let us look closer at Algorithm 1. First, several variables are declared:

- $N$ : static list of neighboring abstract agents, associated with the weights of the respective edges.
- $\tilde{N}$ : dynamic list of neighbors, initially  $N$  with the addition of  $s$ . The list shortens as the neighbors indicate that they have joined other agents.
- $C$ : agents that compose the list of components of  $s$ , where  $s$  is a leader of a group of agents.  $C$  is composed of triplets of type

---

**Algorithm 2:** SELECT-LEADER( $C, C', level$ ): Deterministic leader selection.

---

**Input** : primary components list  $C$ , secondary components list  $C', level$   
**Output**: new leader leader, possibly updated  $C$   
**Locals** : max-size  $\leftarrow -1, Best-Nodes \leftarrow NIL$

```

1 forall ( $t, N', 0$ )  $\in C \cup C'$  do
2   size  $\leftarrow |(\{t\} \times N') \cap ((C \times C') \cup (C' \times C))|$ 
3   if size > max-size then
4     Best-Nodes  $\leftarrow$  NIL
5     max-size  $\leftarrow$  size
6   if size = max-size then
7     Best-Nodes  $\leftarrow$  Best-Nodes  $\cup \{t\}$ 
8 leader  $\leftarrow$  DETERMINISTIC-PICK(Best-Nodes)
9 if (leader,  $\cdot, level_{ldr}$ )  $\in C$  then
10  levelldr  $\leftarrow$  level

```

---

( $a, N_a, level$ ). A value  $level > 0$  indicates that the simple agent  $a$  was selected as a leader at that level.  $N_a$  is the list of  $a$ 's neighbors in the constraints graph (neighbors at level 0, composed of single agents).

- leader: the leader of an agent  $s$  in the groups hierarchy. The value USER indicates that  $s$  represents the top-level group, and that the solutions which it produces are received by the user.
- $N_{ldr}$ : list of neighbors of an agent that is a leader. This list grows as the neighbors at the current level indicate that they have joined with another agent; it contains the leaders of these neighbors.
- parent: the parent of the leaf-level simple agent. For example, in Figure 2 the parent of  $f$  is  $j$ , but the leader of (the representative agent)  $f$  is  $i$ .
- pairs: list of edges between agents that are members of two subgroups that have been merged, ordered by increasing weights of the edges. Pairs are composed of members of *different* subgroups.

The communication procedures used in the algorithm also deserve an explanation. Since at a given level we are only interested in receiving messages from agents that are at the same level, each message is tagged with a level parameter:

- SEND( $destination, tag, message$ ) — sends message with a given tag; never blocks.
- RECEIVE( $tag$ ) — reads only messages with the given tag attached, and leaves other messages in the queue; blocks if there are no appropriate messages.

All messages are tagged with the level variable. As a result, the tag parameter is omitted in the listings of the procedures, and is assumed to be equal to level. In other words, messages are always tagged with the agent's current level variable value, and only messages whose tag equals to level are extracted in RECEIVE(). The resulting code looks similar to the common send/receive primitives, but the reader should be aware of their different semantics.

During the algorithm's execution, the following message types are used:

- Join( $sender$ ): a request to join an agent. If the receiving agent sends a Join as well, an agreement on joining is reached.
- NoJoin: a negative answer to a Join request.
- Components( $components$ ): after agreement on joining has been reached, the agents use this message in order to inform each other about their components sets.
- Done( $sender, leader$ ): a message from a neighbor notifying that it

State	Action
Receive	Initial state. Wait for an appropriately tagged message. Upon receiving such message, dispatch to the corresponding state.
Join_g	Send components to g, and go to Receive.
Join_t	Send NoJoin to t, and go to Receive.
NoJoin	Send Join to chosen agent, and go to Receive.
Components	Send Done messages to all neighbors, and go to Receive.
Done	If the updated dynamic neighbors list is empty, send Leader message to the representative agent. Go to Receive.
Leader	If the reset neighbors list is empty, send Search messages to all agents and go to Receive. Otherwise, go to NoJoin.
Search	Final state.

**Table 1.** Control flow of Algorithm 1.

has performed a join, and which agent has been designated as its leader.

- **Leader**(neighbors, components, single, level): one of two messages (sent by the merging agents) which notify the receiving agent that it has been selected as a leader at a given level. The receiving agent merges the received lists of neighbors and components with its own lists, to get the resulting neighbors and components lists. If single is TRUE, there is only one incoming Leader message, and the agent “merges” with itself, without the selection of a leader. The very first Leader messages which all agents send to themselves bootstrap the algorithm.
- **Search**: originates from the top-level agent, and indicates that the partitioning is over. This message begins the search phase of DESRS.

An informal proof of the correctness of the partition algorithm (Algorithm 1) can be constructed as follows: show that Algorithm 1 results in a correct partitioning of agents into a binary tree of groups (as shown in the example in Figure 2).

The control flow of the algorithm, in a given agent, can be modeled as shown in Table 1. With the help of this model, there are some invariants that can be observed in the group partitioning algorithm’s flow. First, an agent will not send a Done message, unless it has found another agent to join with (which will be the same agent in the extreme case). Also, an agent will not send a Leader message to the representative agent of the new group, unless it has received the Done messages from all its neighbors. Since an agent can only change its level in two cases: when sending a Leader message, and when receiving a Leader message, it is not possible for an agent s to deadlock waiting for an answer to a Join request from an agent t. This is true because t will not change its level before receiving a Done message from s (since s is one of t’s neighbors).

Moreover, each agent will eventually find an agent to join with, since it arbitrarily picks an agent to try and join with from agents to which it is connected with a minimal-weight constraint. Since a cycle with ever-shrinking weights is not possible, there must exist a pair of agents, each of which is connected to the other via a minimal-weight constraint, and thus they will pick each other after a finite number of attempts during the joining process. After this happens, these agents are removed from the dynamic neighbors lists of their respective neighbors, and this argument can be re-applied.

Therefore, in each level all representative agents at that level will pass through the following events: receiving a Leader message; sending a finite number of Join requests, and receiving a NoJoin answer for each of their requests in a finite time; receiving a Join answer from some agent g; exchanging Components messages with g; send-

ing Done messages to all neighbors; receiving Done messages from all neighbors, and sending a Leader message. This establishes termination for Algorithm 1.

The resulting partition is also a correct binary tree. First, each representative agent is part of the components list which it represents. Also, an agent receives either two Leader messages with single = FALSE from two different agents, with different components sets, and becomes a new node in the binary partitioning tree. Or, it receives one Leader message from itself with single = TRUE (which happens when an agent does not succeed in joining another agent), and in this case no new node is formed (level is incremented by 1, the components set remains the same, and the neighbors list is updated).

The binary partitioning tree is essentially built bottom-up, resulting in a tree similar to Figure 2. This establishes correctness for Algorithm 1.

### 3 Search for solutions

In the search phase of DESRS, presented in Algorithm 3, all primitive agents independently initiate empty partial assignments (PAs), which flow up and down in the hierarchy tree formed during the group-partitioning phase (Algorithm 1). The growing partial assignments are distinguished by IDs, assigned to them during their initialization as empty PAs.

Each primitive agent which receives an Assignment message, attempts to combine it with compatible values from its own domain, and send it further. When no compatible value exists, a Nogood message with a resolved explanation is sent to the “culprit” agent, which is determined during the resolution process.

The precise flow of messages is as follows. An Assignment message received by a primitive agent is combined to a value of that agent and sent up, as described above. A representative agent receiving such a message from one of its child agents sends it either up in the hierarchy (if the PA covers the whole component), or to the other child.

When an Assignment message is sent down in the partition hierarchy, it is continuously forwarded down until it reaches a primitive agent. Each representative agent randomly decides, to which child agent the message will be sent.

Nogood messages are sent among primitive agents, in a direction opposite to the partial assignments growth. When a primitive agent discovers that a value in its domain is incompatible with a given PA, the chronologically first conflicting agent in the PA is recorded as the explanation for this failure. When the domain is emptied, these explanations are united with explanations received in Nogood messages (for the same PA), and the most recent agent in the combined ex-

---

**Algorithm 3:** DESRS-SOLVE(s): Searching for solutions.

---

**Input** : agent  $s$ , output from Algorithm 1, domain  $D$ , child agents  $c_{0,1}$ , primitive child indicators  $\text{prim}_{0,1}$   
**Output**: a global solution is sent to USER  
**Locals** :  $\text{Id-Map}[\cdot]$

```
1 loop forever do
2   switch RECEIVE() do
3     ▷ Continuing Algorithm 1...
4   case Search
5     level  $\leftarrow s$ 
6     SEND( $s$ , Assignment( $s$ ,  $s$ , NIL, TRUE))
7   case Assignment( $t$ ,  $\text{id}$ , PA, primitive)
8     if primitive then
9        $\text{Id-Map}[\text{id}] \leftarrow \langle \text{PA}, D, \emptyset \rangle$ 
10      else if  $\exists i : t = c_i$  then
11        if  $c_{1-i} \in \text{PA}$  then
12          SEND( $\text{leader}$ , Assignment( $s$ ,  $\text{id}$ , PA, FALSE))
13        else
14          SEND( $c_{1-i}$ , Assignment( $s$ ,  $\text{id}$ , PA,  $\text{prim}_{1-i}$ ))
15      else
16         $i \leftarrow \text{RANDOM}(\{0,1\})$ 
17        SEND( $c_i$ , Assignment( $s$ ,  $\text{id}$ , PA,  $\text{prim}_i$ ))
18   case Nogood( $\text{id}$ ,  $\text{exp}$ )
19      $\langle \cdot, \cdot, \text{united-exp} \rangle \leftarrow \text{Id-Map}[\text{id}]$ 
20      $\text{united-exp} \leftarrow \text{united-exp} \cup \text{exp}$ 
21   case Assignment( $\cdot$ ,  $\text{id}$ ,  $\cdot$ , TRUE)  $\vee$  Nogood( $\text{id}$ ,  $\cdot$ )
22      $\langle \text{PA}, \text{Values}, \text{exp} \rangle \leftarrow \text{Id-Map}[\text{id}]$ 
23      $v \leftarrow \text{NIL}$ 
24     while  $v = \text{NIL} \wedge \text{Values} \neq \emptyset$  do
25        $v \leftarrow \text{RANDOM}(\text{Values})$ 
26        $\text{Values} \leftarrow \text{Values} \setminus \{v\}$ 
27       for  $(r = w) \in \text{PA}$  (left-to-right, neighbors of  $s$  only)
28         do
29           if CHECK( $v$ ,  $r$ ,  $w$ ) then
30              $\text{exp} \leftarrow \text{exp} \cup \{r\}$ 
31              $v \leftarrow \text{NIL}$ 
32             break
33       if  $v \neq \text{NIL}$  then
34         SEND( $\text{parent}$ , Assignment( $s$ ,  $\text{id}$ ,  $\langle \text{PA}, (s = v) \rangle$ , FALSE))
35       else if  $\text{exp} \neq \emptyset$  then
36         for  $r \in \text{PA}$  (right-to-left) do
37           if  $r \in \text{exp}$  then
38             SEND( $r$ , Nogood( $\text{id}$ ,  $\text{exp} \setminus \{r\}$ ))
39             break
40       else
41         SEND(USER, Nogood( $\text{id}$ ,  $\text{exp}$ ))
```

---

planation is designated as the “culprit”. The united explanation (excluding the “culprit”) is then sent to this agent in a Nogood message [2, 3, 13].

In a problem with  $n$  agents,  $n$  independent partial assignments can be grown, with *Nogoods* back-jumping to failure culprits. The first PA which grows to the full solution, or results in an empty *Nogood*, ends the search process, which is reminiscent of CONCDB [13].

During the execution of the algorithm, the following message types are in use:

- Assignment( $t$ ,  $\text{id}$ , PA, primitive): a partial solution which is sent by agent  $t$  up in the partition hierarchy. The  $\text{id}$  is unique for the growing PA, which contains the ordered partial assignment. The boolean field primitive indicates whether the message has been sent to a primitive agent (a leaf in the group partitioning hierarchy), or its role as a representative agent (non-leaf).

- Nogood( $\text{id}$ ,  $\text{exp}$ ): a resolved nogood sent to the “culprit” agent. The  $\text{id}$  field is equal to the  $\text{id}$  in the Assignment messages with the corresponding (inconsistent) PA, and  $\text{exp}$  is the resolved explanation, which in DESRS is a set of agents.

Algorithm 1 has been implicitly modified to produce the following additional output for each representative agent:

- $c_0, c_1$ : child agents.
- $\text{prim}_0, \text{prim}_1$ : whether the corresponding child agent is a primitive agent (a leaf in the hierarchy).

Note that for solving by DESRS, GROUP-PARTITION() need not produce pairs (the list of constraints between the two sub-components).

A map  $\text{Id-Map}$  is maintained in each primitive agent, holding mappings from  $\text{id}$ s to tuples of the form  $\langle \text{PA}, \text{values}, \text{exp} \rangle$ . Here, PA is the partial assignment, as it was received from the parent agent, values is the current untried subset of values in the current domain, and  $\text{exp}$  is the growing explanation, which will be used if and when values becomes empty.

## 4 Experimental evaluation

Experimental evaluation of the DESRS algorithm has been conducted using an asynchronous simulator. To simulate asynchronous agents, the simulator implements agents as Java threads. Threads (agents) run asynchronously, exchanging messages. After the algorithm is initiated, agents block on incoming message queues and become active when messages are received.

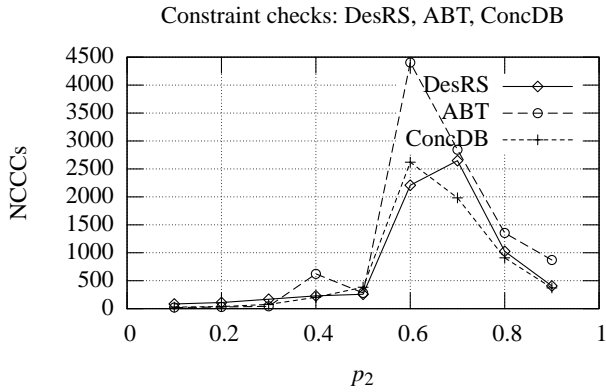
Experiments were conducted on random networks of constraints. The network of constraints, in each of the experiments, is generated randomly by selecting the probability  $p_1$  of a constraint among any pair of variables (*constraint density*) and the probability  $p_2$ , for the occurrence of a violation among two assignments of values to a constrained pair of variables (*constraint tightness*) [8, 9].

Figure 3 compares DESRS to ABT on a set of randomly generated problems of moderate complexity ( $n = 10$ ,  $|D| = 10$ ,  $p_1 = 0.5$ ). Figure 4 compares the same algorithms on a set of hard random problems ( $n = 20$ ,  $|D| = 10$ ,  $p_1 = 0.4$ ).

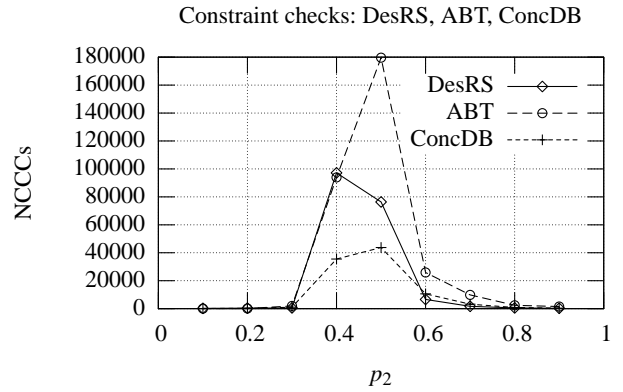
We see that DESRS outperforms ABT on all problems. This is true with respect to both measures of performance — non-concurrent constraint checks [6] and total number of messages. DESRS performs half the number of NCCCs than ABT for the hardest problem instances with  $n = 10$  agents. The same is true for problems with  $n = 20$  agents (Figure 4). The same advantage of a factor of two, for DESRS over ABT, holds for the total number of messages sent (e.g., the network load).

The DESRS algorithm uses multiple search processes to scan the search space concurrently. Another DISCSP search algorithm also uses multiple search processes. CONCDB [13] generates search processes dynamically during search. Figures 3, 4 present also the results of comparing DESRS to CONCDB. The performance of DESRS is similar to CONCDB on the  $n = 10$  agents problems with respect to the number of non-concurrent constraint checks (Figure 3(a)) [6]. On the other hand, DESRS performs about twice the number of constraint checks of CONCDB in the phase transition region of the  $n = 20$  problems set (Figure 4(a)), and uses more messages than CONCDB in all problems.

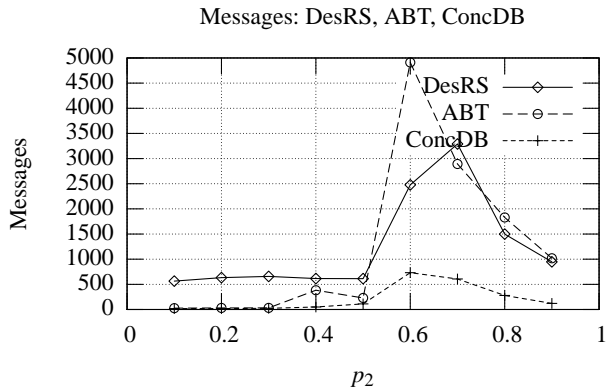
It is interesting to investigate the impact of the partitioning heuristic on the performance of the search algorithm. Figure 5 presents a comparison of DESRS against ANTIDESRS — algorithm similar to



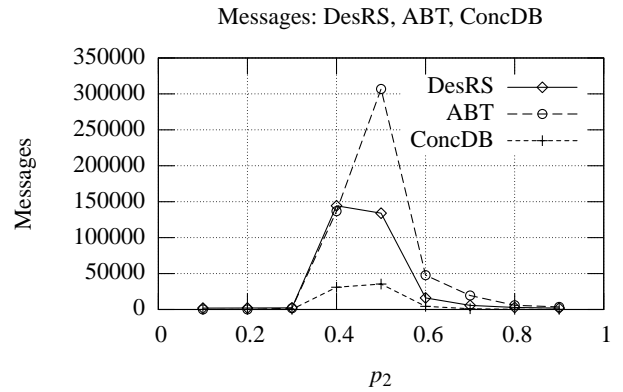
(a) Non-concurrent constraint checks.



(a) Non-concurrent constraint checks.



(b) Total number of messages.



(b) Total number of messages.

**Figure 3.** Comparing DESRS with ABT and CONCDB on random problems with 10 agents, domain size of 10, and  $p_1 = 0.5$ .

**Figure 4.** Comparing DESRS with ABT and CONCDB on random problems with 20 agents, domain size of 10, and  $p_1 = 0.4$ .

DESRS, but with the weights controlling the partitioning reversed (during the groups partitioning phase). That is, instead of pushing hard constraints down in the hierarchy, they are pushed up during the execution of GROUP-PARTITION().

It is easy to see that the difference, especially in terms of constraint checks, is large — good partitioning hierarchy is vital to the performance of DESRS.<sup>3</sup>

## 5 Discussion

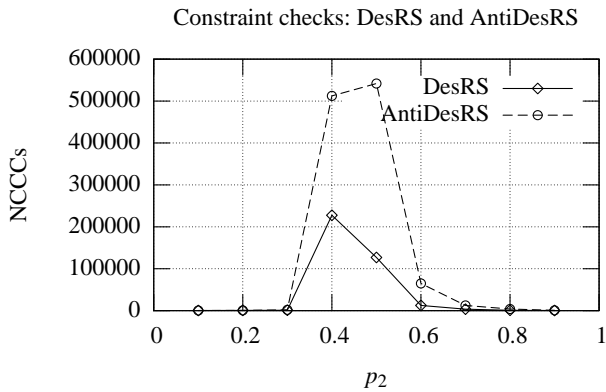
A new search algorithm for distributed constraint problems (DISCSPs) is presented. The new algorithm uses a hierarchy of groups of agents to partition the problem. Solutions are generated by multiple concurrent search processes, all coordinated on the hierarchy of groups. Multiple agents initialize partial solutions concurrently and send them to group leaders. Leaders of groups of agents coordinate the generation of each solution by determining the groups that are merged with partial solutions, to form larger solutions.

<sup>3</sup> The results for DESRS differ from those in Figure 4 due to a different experimental environment.

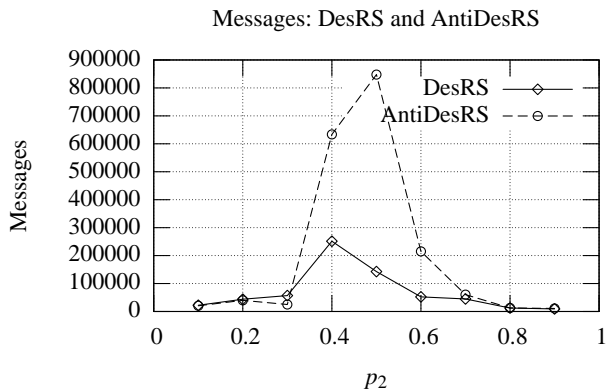
One can think of the hierarchical structure of groups as a means of defining a partial order of search for the multiple search processes. The resulting algorithm performs better than asynchronous backtracking (ABT). It performs half the number of non-concurrent constraints checks than ABT for hard instances of randomly generated DISCSPs. It also sends half the number of messages than ABT for the same problems.

Comparing DESRS to the best performing concurrent search algorithm, CONCDB, is less successful. For random DISCSPs with 20 agents, the run-time of DESRS is longer than that of CONCDB. However, one has to bear in mind that CONCDB takes advantage of communication among search processes. Specifically, such communication is used in terminating processes because of dead-ends discovered by other search processes [13]. This option is in principle possible also for DESRS. Leaders of groups could inform the search processes passing through them about discovered *Nogoods* and terminate some of the partial assignments. This potential improvement of DESRS is left for a future study.

*Asynchronous Partial Overlay* (APO) is a recently introduced algorithm for solving DISCSPs [4]. It uses a method of *cooperative mediation*. In APO, some of the agents act as mediators. Mediators



(a) Non-concurrent constraint checks.



(b) Total number of messages.

**Figure 5.** Comparing DESRS with ANTIDESRS on random problems with 20 agents, domain size of 10, and  $p_1 = 0.4$ .

construct small overlapping portions of the constraints network and solve them. As the problem solving unfolds, mediators increase the size of their subproblems [4]. The approach taken by the DESRS algorithm in the present paper is principally different. The proposed hierarchical search algorithm uses a distributed computation in order to generate a good partition of the overall problem. We have shown experimentally that the group partitioning phase is critical for the performance of DESRS. However, partitioning is completely independent from the search phase. Moreover, agents in hierarchical search do not centralize subproblems. Instead, partial assignments are forwarded to higher levels in the partition hierarchy.

*Distributed Pseudotree Optimization* (DPOP) is a recent algorithm for DISCSP optimization [7]. DPOP builds a depth-first traversal tree from an elected leader. It subsequently incrementally computes all partial solutions by propagating *utility* and *value* messages in the resulting tree. In DPOP, the DFS and the optimization phases are not independent: absence of constraint links between the tree branches is a requirement of the algorithm. Number of messages is linear, but their size is exponential in the *induced width* of the problem. The dynamic programming approach taken by DPOP is very different from DESRS.

Finally, the hierarchy of groups can serve other purposes, unrelated to distributed constraint solving. The group partitioning phase is independent of the search phase. The input to the former is specified by a connected undirected graph of agents with weighted connections (edges), and unrestricted communication between agents. A method of combining weights of several edges between groups of agents is a part of the input. The output of the group partitioning is specified by the established hierarchy of groups, where some of the agents are designated as representative agents (leaders). This hierarchy has a bias towards connections (primitive of aggregative, combining several edges) with high weight remaining at the low levels of the hierarchy.

The process of partitioning into a hierarchy of groups can be applied to other domains, where connectivity plays a primary role. An example of such a domain is the area of social networks [1]. Consider a network of people, where mutual ties are represented by edges with a weight in some interval. These edges can, for example, represent amounts of phone conversations between pairs of people. Such a network will exhibit the properties identified above as necessary for Algorithm 1. Moreover, since the algorithm is concurrent and consumes few resources, it can be scheduled to run on all agents as frequently as necessary in order to accommodate social network updates. Interesting applications of resulting hierarchies and emerging group leaders can be considered.

## REFERENCES

- [1] John Barnes, 'Class and committees in a Norwegian island parish', *Human Relations*, 7, 39–58, (February 1954).
- [2] Christian Bessière, Arnold Maestre, Ismel Brito, and Pedro Meseguer, 'Asynchronous backtracking without adding links: A new member in the ABT family', *Artificial Intelligence*, 161(1–2), 7–24, (January 2005).
- [3] Matthew Ginsberg, 'Dynamic backtracking', *Artificial Intelligence Research*, 1, 25–46, (August 1993).
- [4] Roger Mailler and Victor Lesser, 'Using cooperative mediation to solve distributed constraint satisfaction problems', in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pp. 446–453, New York, New York, USA, (August 2004).
- [5] Amnon Meisels, 'Distributed constraints: Algorithms, performance, communication', in *CP-2004: Tutorials*, Toronto, Canada, (September 2004).
- [6] Amnon Meisels, Eliezer Kaplansky, Igor Razgon, and Roie Zivan, 'Comparing performance of distributed constraints processing algorithms', in *Proceedings of the Third Workshop on Distributed Constraint Reasoning*, pp. 86–93, Bologna, Italy, (July 2002).
- [7] Adrian Petcu and Boi Faltings, 'A scalable method for multiagent constraint optimization', in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 266–271, Edinburgh, Scotland, (August 2005).
- [8] Patrick Prosser, 'An empirical study of phase transitions in binary constraint satisfaction problems', *Artificial Intelligence*, 81(1–2), 81–109, (March 1996).
- [9] Barbara Smith and Martin Dyer, 'Locating the phase transition in binary constraint satisfaction problems', *Artificial Intelligence*, 81(1–2), 155–181, (March 1996).
- [10] Makoto Yokoo and Katsutoshi Hirayama, 'Algorithms for distributed constraint satisfaction: A review', *Autonomous Agents and Multi-Agent Systems*, 3(2), 185–207, (June 2000).
- [11] Roie Zivan and Amnon Meisels, 'Concurrent backtrack search on DisCSPs', in *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Symposium Conference*, pp. 776–781, Miami Beach, Florida, USA, (May 2004).
- [12] Roie Zivan and Amnon Meisels, 'Concurrent dynamic backtracking for distributed CSPs', in *Principles and Practice of Constraint Programming — CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pp. 782–787, Toronto, Canada, (January 2004).
- [13] Roie Zivan and Amnon Meisels, 'Concurrent search for distributed CSPs', *Artificial Intelligence Journal*, 170(4–5), 440–461, (April 2006).