Ben-Gurion University of the Negev

Faculty of Natural Sciences

Department of Computer Science

Hierarchical Search on DisCSPs

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

Michael Orlov (orlovm@cs.bgu.ac.il)

July 2006

Subject:	Hierarchical Search on DisCSPs
Author:	Michael Orlov
Advisor:	Prof. Amnon Meisels
Department:	Computer Science
Faculty:	Natural Sciences
Organization:	Ben-Gurion University of the Negev

Signatures:

Author: Michael Orlov	Date
Advisor: Prof. Amnon Meisels	Date
Dept. Committee Chairman	Date

Preface

Two new distributed search algorithms for Distributed CSPs are proposed.

In one algorithm, search is performed concurrently on disjoint parts of the global search space, by agents that constitute these parts. Agents form a hierarchy of groups and each group generates consistent partial solutions. Partial solutions are produced concurrently and are combined into consistent global solutions by agents that are higher in the hierarchy.

In another algorithm, concurrent independent backtracking search processes grow partial assignments along a hierarchy of agent groups, with each agent participating in multiple search processes. Stochastic choices for the order of assigning agents are taken after backtracking, securing the growing partial assignments from stalling in local minima.

The first part of both algorithms partitions the distributed constraint network into a binary tree of groups. After completing the partition, the hierarchy of groups starts to produce partial solutions. The distributed partition algorithm uses a heuristic that selects to join neighbors that are strongly constrained, into groups. This is done concurrently at all levels of the hierarchy. The result is a new and special effective order of assigning agents, for both algorithms.

In DISHS, partial solutions produced at one level are combined into larger partial solutions by agents at the next higher level. The process stops when the top-level agent, the leader of all groups, reports either a solution or a failure.

In DESRS, 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*.

Acknowledgements

I would like to express my utmost gratitude to my advisor, Prof. Amnon Meisels, for the scientific and moral support throughout our work and research.

I would also like to thank Marina for enduring this long period of my life. :)

Contents

Chapter 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) [4, 25], 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 [25].

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 [31–33]. CONCDB maintains a dynamic number of backtracking search processes and generates an efficient concurrent performance [33].

The present study 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 *distributed hierarchical search (*DISHS*)* 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 both hierarchical search algorithms is the smart partition of the agents in the hierarchy of groups.

Hierarchical search 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. Solutions are generated within groups and become larger by merging partial solutions. In the second phase of distributed hierarchical search, 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. In the DISHS algorithm group

solutions are generated by the group leaders. Agents that are leaders combine at most two partial solutions. In the DESRS algorithm solutions are generated by passing compatible partial solutions among all agents. Here too, the grouping results in a partial order among agents which incrementally generate the solutions.

Two forms of hierarchical search (HS) are presented in this study. In one version, partial solutions are combined by the leaders of each group. The leaders check consistency of their combined partial solutions by communicating with constraining agents in other groups. In general, agents perform two tasks concurrently. Each agent responds to constraints checking messages from leaders of its enclosing groups, checking its constraints with other agents. Agents that are leaders combine partial solutions into larger ones.

The other version of distributed hierarchical search 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. This form of hierarchical search is called *Descending Requirements Search* (DESRS). Requirements for extending partial solutions are being *routed down* (i.e., descended) by the leaders of groups.

Hierarchical 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. By maintaining multiple search processes concurrently, search can be made more efficient, especially if multiple solutions are needed.

Descending requirements search maintains multiple concurrent search procedures, that are coordinated within each group (level) in the hierarchy by the leader of that group.

Both algorithms are evaluated experimentally on randomly generated DISCSPs [15–17]. They are compared to existing search algorithms — ABT [25], CONCDB [33], and the DESRS algorithm is shown experimentally to perform better than ABT on all problems, and to perform similar to CONCDB on problems of limited complexity.

Chapter 2 presents CSPs, DISCSPs and distributed search algorithms for DISCSPs. Chapter 3 presents an overview of hierarchical search, including a theoretical treatment of constraint checks bounds. Chapter 4 describes the first phase of hierarchical search — partitioning of agents into a hierarchy of groups. The distributed partitioning algorithm is of interest by itself. This is discussed in short in Chapter 7. Chapter 5 describes the solving phase of DISHS. Chapter 6 describes the requirements-based solving phase, the Descending Requirements Search. In Chapter 7, possible applications of the group partitioning algorithm are discussed. Chapter 8 presents our conclusions.

Chapter 2

Background

2.1 Constraint Satisfaction Problems (CSPs)

Constraint networks have been introduced two decades ago, and have been used to represent a wide range of problems, from scheduling problems to VLSI layout design. Applying constraint processing to timetabling problems, from scheduling courses at the university to staffing jobs and shifts in modern work places, is an interesting set of applications [1, 11, 22]. All of these problems share a common representation which is very general — *constraint networks* (CNs), or *constraint satisfaction problems* (CSPs) [7, 8, 21]. A field of growing interest is that of *distributed constraints satisfaction problems* (DISCSP).

Constraints satisfaction problems (CSPs) have been studied for many years. CSP is a well known NP-complete problem, and its implementations are often used in order to solve problems such as timetabling. A CSP can be viewed as a tuple $\langle X, D, C \rangle$, where X is a finite set of variables $x_1, x_2, \ldots, x_m$, and D is a set of domains $D_1, D_2, \ldots, D_m$. Each domain D_i contains the finite set of values which can be assigned to variable X_i. C is a set of relations (constraints) that specify for each value v_j in D_i, in what cases it cannot be assigned to variable X_i. The most commonly used constraints are binary constraints which define for every two variables x_i and x_j a subset of the Cartesian product of their domains $D_i \times D_j$. In that sense, constraints define the allowed pairs of values among every pair of constrained variables. A solution to the CSP is an assignment of a value from each variable's domain to the respective variable, that does not violate any constraints from C [8, 21].

In standard (centralized) CSPs, all the problem's data — including variables, domains and constraints, are accessible by a single computer (agent) that searches for a consistent solution to the problem. A search on a CSP that does not contain such a solution ends by declaring the CSP as unsolvable.

As an example of a CSP, one can take the currently popular Sudoku puzzles. The following definition is given in [18].

Definition 1. A Sudoku square of order n consists of n^4 variables formed into a $n^2 \times n^2$ grid with values from 1 to n^2 such that the entries in each row, each column and in each of the n^2 major $n \times n$ blocks are *alldifferent*.

			7	5				
	3			4	8		2	
1								6
	4							8
7	9						3	1
2							7	
5								7
	8		3	2			4	
				6	9			

Figure 2.1: A Sudoku puzzle example (rated at easy-medium difficulty level).

Figure 2.1 shows an instance of such puzzle, where the purpose is to find a consistent Sudoku square of order 3. This instance can be formally described as a tuple $\langle X, D, C \rangle$, where $X = \{x_{1,1}, \ldots, x_{9,9}\}$ is a set of 81 variables which correspond to the 81 cells in the grid, and $D = \{D_{1,1}, \ldots, D_{9,9}\}$ is a set of domains, where the domains of the empty cells equal $\{1, \ldots, 9\}$, and the domains of the filled cells are singletons with just the given number. C contains the 27 *alldifferent* constraints, as described in Definition 1. Note that the constraints are not binary.

Algorithm evaluation is in many cases performed on randomly generated binary constraint satisfaction problems [17]. The two main parameters used in problem generation are *constraint density* p_1 and *constraint tightness* p_2. Constraint density is the probability of a constraint between two variables, and constraint

tightness is the probability of a conflicting pair of values in a given constraint.[1]

2.2 Distributed CSPs

Distributed constraint satisfaction problems (DISCSPs) are composed of agents, each holding its local constraint network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [20, 23]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages with other agents, to check consistency of their proposed assignments against constraints with variables owned by different agents. Following common practice, it is assumed that an agent can send messages to any one of the other agents [4, 6, 23].

Distributed CSPs are an elegant model for many every day combinatorial problems that are distributed by nature. Take for example a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint satisfaction problem for each ward. Some of the nurses in each ward are qualified to work in the emergency room. Hospital regulations require a certain number of qualified nurses (e.g. for emergency room) in each shift. This imposes constraints among the timetables of different wards. Assigning an unqualified nurse to some shift is acceptable only if there are enough qualified nurses assigned to that shift in the other wards. This results in a complex Distributed CSP [11, 12, 20].

A search procedure for a consistent assignment of all agents in a DISCSP is a distributed algorithm. All agents cooperate in the search for a globally consistent solution. The distributed nature of a search algorithm on DISCSPs produces many additional features to those of standard CSPs. The correctness and termination conditions are typically more complex on DISCSPs than in standard backtracking search like CSP [24, 25]. The appropriate performance measures for distributed constrained search must take concurrency into account [13, 16]. Since agents in distributed search algorithms communicate by message passing, the impact of message delay on the performance of different algorithms must also be investigated. These are just few features that come up in DISCSPs.

The solution involves assignments of all agents to all their variables and exchange of information among all agents, to check the consistency of assignments with constraints among agents. The *state* of the search at a specific step is defined by the current assignments held by agents for their variables. In some algorithms such as *Asynchronous Backtracking* [23] and *Distributed Breakout* [29], all agents hold assignments during the search while in others such as *Synchronous Backtracking* [23] the state is incrementally constructed and some of the agents do not hold

[1] These definitions deviate slightly from Prosser [17], who defines p_1 and p_2 as the proportion of constraints in the graph, and the proportion of conflicting value pairs in each constraint, respectively, and not as probabilities.

an assignment for their variables through some parts of the search. In concurrent search agents hold a varying number of multiple assignments, as part of multiple search processes [31, 32]. The present study includes two versions of hierarchical search and agents hold a varying number of assignments.

In a *Distributed CSP* (DISCSP), the variables of a CSP are distributed among a set of agents A. each variable x_i of X is assigned to an agent a_j which is one of the agents in set A. We refer to a variable x_i of a_j as *owned* by the agent A_j. In DISCSPs the constraints set C contains two types of constraints:

1. *Intra constraints*: constraints between variables, owned by the same agent.

2. *Inter constraints*: constraints between variables, owned by different agents.

Intra constraints are solved by each agent locally, using centralized CSP methods. Inter constraints must be solved via some cooperation between the constrained agents. In order for two agents a_i and a_j, which hold variables with constrained values in their domains, to assign values to their variables, at least one of them has to find out if the assignment value it chose for its variable is consistent with the other agent's assignment. This can be done only if the agent a_i holding the constraint (without loss of generality) receives a message from agent a_j that contains a_j's assignment, and finds an assignment value for its variable, consistent with the assignment of a_j. This means that in order to find a consistent assignment for agents with constrained variables, one of the following properties must hold:

1. The agents' order of search assures that the agent holding the constraint will be obliged to find an assignment consistent with the assignment of the agent not holding the constraint.

2. Every agent whose variable is involved in some constraint $R \in C$, always holds the constraint R.

Following all former studies on DISCSPs, agents communicate by sending messages. The delay of each message is assumed to be finite. The content of messages is not homogeneous in all algorithms. In the most common approach, messages contain assignments of values to agents' variables. Agents can send messages to all constraining agents [4, 23]. In most studies a relaxed assumption is used, that all agents can communicate with all other agents. In some algorithms agents send messages only to agents which are their neighbors in the constraints network (agents they have inter constraints with) [4, 23].

In most algorithms for solving DISCSPs there are two basic types of messages:

1. Messages that are sent by agents that have succeeded to find an assignment for their local constraints network and ask other agents to check inter constraints against the assignment received (*ok?* message [28], *info_val* message [10]).

2. Messages that are sent by agents that fail to find an assignment for their local constraints network, that is consistent with their view of other agents' assignments. These messages essentially request other agents to replace their current assignment (*Nogood* message [28], backtrack *bt* message [10]).

A solution to a Distributed CSP is a function that assigns a single value from each variable's domain to the variable. A "no solution" declaration is returned for DISCSPs with no solution to satisfy the constraints in C.

Most studies on DISCSPs restrict themselves to DISCSPs in which each agent owns exactly one variable. This choice simplifies the presentation of the algorithms and has two major justifications:

- An agent holding k variables can be simulated as k virtual agents.
- The agent's k variables can be replaced by one variable whose domain includes all the consistent tuples of assignments for the k replaced variables.

2.3 Asynchronous Backtracking

The *Asynchronous Backtracking* algorithm (ABT), first presented by Yokoo et al. [27], was constructed to remove the drawbacks of *Synchronous Backtracking* (simple chronological backtracking) algorithm by allowing agents to run concurrently and asynchronously. In all variants of ABT, the algorithm is presented for DISCSPs in which each agent holds exactly one variable. Each agent assigns its variable, and communicates the assignment it made to the relevant agents. In the ABT algorithm a total order of priorities is defined among agents, and therefore for each binary constraint only the agent with the lower priority needs to hold the constraint. A link in the constraints network is *directed* from the agent with the higher priority to the agent with the lower priority among the two constrained agents.

Agents instantiate their variables concurrently, and send their assigned value to the agents that are connected to them by outgoing links. After that, the agents wait for and respond to messages.

After each update of its assignment, an agent sends its new assignment through all outgoing links. An agent which receives an assignment (the lower priority agent of the link) tries to find an assignment to its variable which does not violate a constraint with the assignment it received.

The *ok?* messages carry an assignment of an agent. When an agent A_i receives an *ok?* message from agent A_j, it places the received assignment in a data structure called *Agent_View*, which holds the last assignment A_i received from higher priority neighbors such as A_j. Then A_i checks if its current assignment is still consistent with its *Agent_View*. If not, A_i searches its domain for a new consistent value. If it finds one, it assigns its variable and sends *ok?* messages to all agents linked with it, with lower priority. Otherwise, A_i backtracks.

The *backtrack* operation is executed by sending a *Nogood* message that contains an inconsistent partial assignment to the agent with the lowest priority among

the agents whose assignments are included in the inconsistent tuple (i.e., the *Nogood*). Agent A_i that sent a *Nogood* message to agent A_j assumes that A_j will change its assignment, therefore A_i removes the assignment of A_j from its *Agent_View*, and makes an attempt to assign its variable a value which is consistent with the refreshed *Agent_View*.

The issue of how to resolve the inconsistent partial assignment (using *Nogood*s sent in backtracking messages) evolved through the different versions of ABT. A shorter *Nogood* would mean backjumping further up in the search-tree, but finding such a short *Nogood* can be wasteful in computational time. In the early versions [26, 27], Yokoo suggests sending the complete *Agent_View* as a Nogood. *Agent_View* might be in many cases not a minimal *Nogood*, i.e., it might contain assignments that if removed, the partial assignment remaining still eliminates all values in the agents domain. In later versions of the ABT algorithm [4], a minimal *Nogood* is resolved using *Dynamic Backtracking* (DB) method [9].

In Dynamic Backtracking, an agent A_i that receives a *Nogood* adds it to its list of constraints. Since the *Nogood* can include assignments of some agent A_j, which A_i was not previously constrained with, A_i, after adding A_j's assignment to its *Agent_View*, sends a message to A_j asking it to add A_i to its list of outgoing links. A_j, after adding the link, sends an *ok?* message to A_i each time it reassigns its variable. After storing the *Nogood*, A_i checks if its assignment is still consistent. If it is, a message is sent to the agent the *Nogood* was received from. This resending of the assignment is crucial, since as mentioned above, the agent sending a *Nogood* assumes that the receiver of the *Nogood* replaces its assignment. If the old assignment is inconsistent, A_i tries to find a new assignment as done when an *ok?* message is received.

The algorithm ends successfully when all the agents are idle (i.e., their assignments are consistent with their respective *Agent_View*s) and no message that will change any agent's *Agent_View* (an *ok*? message) or add to agent's constraints (a *Nogood* message) is yet to be received by any agent in the DISCSP. In such a case the assignments held by the agents represent a solution to the DISCSP. The algorithm fails if some agent creates an empty *Nogood*.

Chapter 3

Distributed Hierarchical Search (DISHS)

3.1 Overview of DISHS

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 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).

The particular type of (binary) hierarchy that is at the center of the present study has no special meaning. It mainly reflects the relative ease of performing a distributed binary partition. More complex partition algorithms could in principle be designed, producing a hierarchy of groups that is not a binary tree.

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, either for checking consistency of assignments (Chapter 5), or to request generation of consistent partial solutions (Chapter 6).

All hierarchical search algorithms first partition 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 4.1). After completing the partition, search for a globally consistent solution is performed by merging partial solutions of groups into solutions of higher level groups. Solutions to the highest level group are global solutions to the DISCSP.

Consider the constraints network in Figure 3.1. Note that there is only one

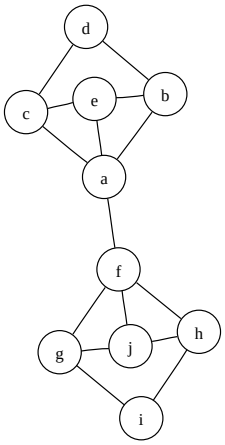


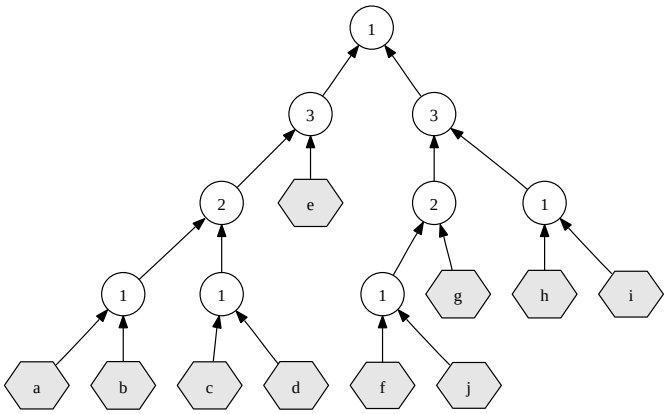
Figure 3.1: Constraints graph with 10 nodes. Edges represent binary constraints.

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 3.2(a), which also specifies the number of constraints between neighboring groups. At level 0 of the tree in Figure 3.2(a) 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.

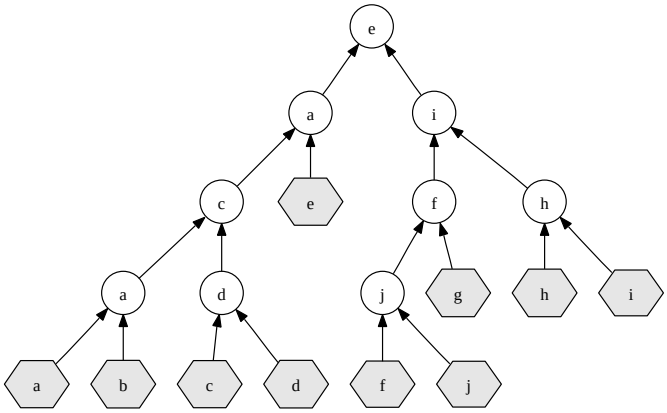
The partitioning process starts with all agents at level 0. Pairs of agents are merged into level-1 groups, and pairs of level-i groups are merged into level $i+1$ groups. If an agent or a group is unable to merge with another group at some level, it becomes a group of one level higher. In this higher 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 3.2(a) 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 shown in Figure 3.2(b). Agent c is the leader of a group with 4 components. It has two subgroups whose leaders are agents a and d. At the solving stage of the DISHS algorithm, agent i generates partial solutions by merging the solutions produced by its two subgroups. In Figure 3.2, the partial solutions are sent to agent i by agent f



(a) Nodes show the number of constraints connecting the merged groups.



(b) A representative agent is responsible for each group.

Figure 3.2: Partitioning of the constraints graph of Figure 3.1.

(which leads $\{f, j, g\}$) and agent h (which leads $\{h, i\}$).

Merging of two partial solutions can be performed either by the leader (algorithm DISHS in Chapter 5), or by the component following a request and a delivery of one of the partial solutions by the leader (algorithm DESRS in Chapter 6).

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 constraint check queries, or to requests for partial solution extensions. Each group representative combines solutions of its children in the partition hierarchy into a consistent assignment for its group. Ultimately, the root node agent produces globally consistent solutions. Chapters 5 and 6 present the two versions of hierarchical search. 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. This approach is presented as the descending requirements search algorithm (DESRS) in Chapter 6.

3.2 Constraint checks bounds

Let us look closer at the partitioning of agents into a binary hierarchy. The main goal of the partition is the parallelization of partial solutions generation. One of the desired properties of the partition is that inconsistencies will be detected as early as possible during the search phase. This helps to reduce the number of inconsistent partial solutions combination attempts in upper levels of the hierarchy. For both of the hierarchical search algorithms, one can intuitively think of this as a heuristic for reducing thrashing. Computationally, this reduces the number of constraint checks that are performed during each combination attempt. To achieve partitions with this feature, one can use a heuristic function to help select the group to merge with. The heuristic that will be used in the present implementation is based on the following definition.

Definition 2. The *weight* of a constraint between two variables is the probability that a random pair of values of these variables is *not in conflict*.

Definition 3. A *virtual constraint* between two groups of variables is the set of all constraints between two agents from both groups.

Definition 4. The *weight of a virtual constraint* between two groups of variables is the probability that a random pair of partial assignments of these groups does not cause any pair of variables from both groups to be in conflict.

The weight is in essence the reverse tightness of a constraint [17]. Consequently, instead of using the number of constraints between group of agents (as was done in Figure 3.2(a)), the weight of a virtual constraint between two groups of agents is approximated by multiplying the weights of the involved constraints.

A representative agent which combines S_1 and S_2 solutions from its child agents, and which has a virtual constraint weight w, will expect to produce wS_1S_2 solutions for its parent in the hierarchy, assuming independence of constraints. Note that the weight is always in the $[0,1]$ range.

In order to minimize the expected number of constraint checks performed during the combining of partial solutions, constraints are ordered by increasing weights.[1] Consider a composite group with k constraints between its child subgroups, with virtual constraint weight w. We now calculate a bound on the expected number of constraint checks performed when combining two partial solutions.

Lemma 3.2.1. *A composite group with k constraints between its child subgroups, with a virtual constraint weight w, the expected number of consecutive constraint checks performed while combining two partial solutions is bounded by*

$$1 + w(k-1) \leq E_k(w) \leq \frac{1-w}{1-\sqrt[k]{w}} \tag{3.2.1}$$

assuming that the constraints are mutually independent, and that the constraint checks are performed in the order of increasing constraint weights.

The upper bound is tight if the constraints have equal weight. The lower bound is tight if $\forall_{1<i\leq k}\ w_i = 1$.

Proof. Let us denote the k constraints composing w as $w_1, \ldots, w_k$. By Definition 2, $w = \prod_{i=1}^{k} w_i$.

Assuming that the constraints are independent, and since the constraint checks stop with the first check that fails, the expected number of constraint checks $E_k(w)$ is given by

$$E_k(w) = \sum_{i=1}^{k} i\left(\prod_{j=1}^{i-1} w_j\right)(1-w_i) + kw = \sum_{i=1}^{k}\prod_{j=1}^{i-1} w_j \tag{3.2.2}$$

where $\left(\prod_{j=1}^{i-1} w_j\right)(1-w_i)$ is the probability that the first $i-1$ constraint checks succeeded, after which the i^{th} check failed.

That is,

$$\begin{aligned} E_k(w) &= (1-w_1) + 2w_1(1-w_2) + \ldots + kw_1\cdots w_{k-1}(1-w_k) && \text{(failure)} \\ &\quad + kw_1\cdots w_k && \text{(success)} \\ &= 1 + w_1 + w_1w_2 + \ldots + w_1\cdots w_{k-1} \end{aligned}$$

Since the weights are ordered by increasing weight, by Lemma 3.2.2 we have:

$$E_k(w) \leq \sum_{i=1}^{k} w^{\frac{i-1}{k}} = \sum_{i=0}^{k-1}\left(\sqrt[k]{w}\right)^i = \frac{1-w}{1-\sqrt[k]{w}}$$

[1] This point becomes quite moot when the constraint checks are not performed by the representative agent itself. However, during the representative agent selection, agents which are incident on a maximal number of constraints are preferred — see Algorithm 4.2.

In case of equal constraint weights, $\prod_{j=1}^{i-1} w_j = w^{\frac{i-1}{k}}$, and the inequality turns into equality, making the upper bound tight.

Noting that $\forall_{1\leq j\leq k-1}\ \prod_{i=1}^{j} w_i \geq w$ establishes the lower bound. In a case where all constraint weights except the first are equal to 1, $w = w_1$ and the lower bound is easily seen to be equal to $E_k(w)$ from (3.2.2). □

Lemma 3.2.2. *Consider* $0 \leq w_1 \leq \ldots \leq w_k \leq 1$ *for* $k \geq 1$, *where* $\prod_{i=1}^{k} w_i = w$. *Then,*

$$\forall_{0\leq j\leq k} \prod_{i=1}^{j} w_i \leq w^{\frac{j}{k}} \tag{3.2.3}$$

Proof. For convenience, we assume that $w_1 > 0$, since the lemma clearly holds for cases where $w = 0$. The proof is by induction on $k \geq 1$. The $k = 1$ case is trivial. Assume that the lemma is correct for all $1 \leq k' \leq k$, and consider the case where $k' = k+1$. For all $0 \leq j \leq k$,

$$\begin{aligned}
\prod_{i=1}^{j} w_i &\leq \left(\frac{w}{w_{k+1}}\right)^{\frac{j}{k}} && \text{(by the induction hypothesis)}\\
&\leq w^{\frac{j}{k}} && \text{(since } 0 < w \leq w_{k+1} \leq 1\text{)}\\
&\leq w^{\frac{j}{k+1}} && \text{(since } 0 \leq w \leq 1\text{)}
\end{aligned}$$

and the subcase where $j = k+1$ is trivial again. □

Since at level l each of the two subgroups has at most 2^{l-1} agents, the number of constraints k at level l is at most 4^{l-1}. If the constraints network is random, with constraints density p_1, the expected number of constraints k at level l is at most $p_1 4^{l-1}$, and the upper bound on the expected number of constraints checks is given by

$$E_l(w) \leq \frac{1-w}{1-w^{\frac{4^{1-l}}{p_1}}} \tag{3.2.4}$$

Note. In the algorithms presented in this study, operations on sets are used in their intuitive meaning. For example, the propositions

$$a \in \{(a,0.2),(b,0.3)\} \qquad\qquad \{(a,N,0)\} \cap \{(a,0.2)\} = \{a\}$$

are considered true. This convention hopefully picks the right trade-off between the algorithms' verbosity and readability.

Chapter 4

Partitioning into a Hierarchy of Groups

Partitioning into groups is a distributed algorithm, such that upon its termination groups of agents are organized into a binary tree. All agents are leafs, and representative agents compose all higher levels of the binary tree (see Figure 3.2). 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 4.1. 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. In the description of the algorithm, the symbol s is used for the agent that selects a neighbor to merge with. This agent (s for self) may represent any group of agents. When a representative agent scans its neighbors for a candidate to merge with, it considers neighbors to any of the group's members.

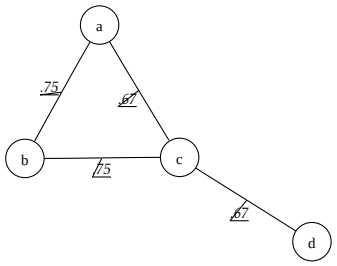
The algorithm works as follows. Each agent s selects one of its neighbors (say, g) as a candidate for merging with. If the selection is mutual, the agents merge. If not, the selected agent g can either respond negatively (i.e., NoJoin) when its own selection was different, or inform s that it has already made a merging decision (i.e., Done$\langle$g, ...$\rangle$). 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.,

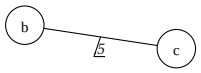
constraints tightness) remains in lower levels of the hierarchy, and inconsistent partial solutions are expected to be pruned as early as possible.

4.1 Partition survey

Before going into the partition algorithm details, let us trace its execution on a small graph-coloring problem. Figure 4.1(a) shows the constraint network, where the domains of a, c and d are $\{1,2,3\}$, and the domain of b is $\{1,2,3,4\}$. Weights on the edges show the percentage of allowed value pairs. For example, the weight of (b,c) edge is 0.75, since three value pairs (equal colors) are forbidden, and therefore the weight is $\frac{4\cdot3-3}{4\cdot3} = 0.75$.



(a) Initial constraint network.



(b) Level 2 of the constraint network partitioning.

Figure 4.1: A small constraint network.

The partitioning process starts when each agent sends a Join message to a minimal-weight neighbor. When several neighbors have minimal weight, one of them is picked arbitrarily. Assume that the messages are: $a \rightarrow c$, $b \rightarrow a$, $c \rightarrow d$, and $d \rightarrow c$. Here, c and d join and send each other components info. Afterwards, they pick a leader, c, and send Done messages to all the neighbors.

Since c sent a Done message to a and b, these agents remove c from their neighbors list. They now join, exchange components info, pick a leader, b, and send a Done message to each other and to c.

Each agent has now received Done messages from all its neighbors, and can send a Leader message to its leader, to activate it at the next level of partitioning. Thus, a and b each send a Leader message to b, and c and d each send a Leader message to c.

At level 2 of the partitioning, b and c join, picking d as the leader (Figure 4.1(b)). The resulting hierarchy of agents is shown in Figure 4.2. In the end, d sends a Search message to all agents in order to initiate the search process.

4.2 Algorithm primitives

Let us look closer at Algorithm 4.1. First, several variables are declared:

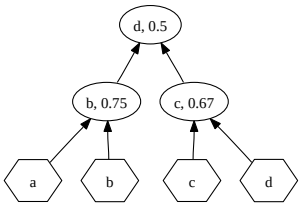


Figure 4.2: The resulting partition hierarchy of the small constraint network, including virtual constraint weights.

- N: static list of neighboring abstract agents, associated with the weights of the respective edges.
- $\tilde{\mathsf{N}}$: 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 $(\mathsf{a}, \mathsf{N_a}, \mathsf{level})$. A value $\mathsf{level} > 0$ indicates that the simple agent a was selected as a leader at that level. $\mathsf{N_a}$ is the list of a's neighbors in the constraints graph (neighbors at level 0).
- 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.
- $\mathsf{N_{ldr}}$: 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 those neighbors.
- parent: the parent of the leaf-level simple agent. For example, in Figure 3.2(b) 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 sub-groups 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 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 (here, DISHS or DESRS).

Note that the algorithms presented here do not require garbage collection of messages — all messages, however tagged, are eventually consumed.

4.3 Detailed description

GROUP-PARTITION() begins with the initialization of the level message tag variable to the special value $\mathfrak{p}$ (partition), which is used exclusively for Leader and Search messages, and sends itself a Leader message (line 1) — effectively performing a **goto** to the last case in the **switch** statement (line 28).

The agent then receives its own Leader message, and initializes the values of its level (initially 0) and its list of neighbors and their weights — N. It initializes a dynamic list of neighbors — $\tilde{\mathsf{N}}$ (including the agent itself, with maximum weight connection), a list of components (i.e., single agents), initially containing just the agent — C — and the leader's neighbors list $\mathsf{N}_{\mathsf{ldr}}$. The list of the leader's neighbors

Algorithm 4.1: GROUP-PARTITION(s, N): Partitioning into groups.

Input : agent s, its neighbors N
Output: components C, pairs, leader, parent ← USER
Locals : Ñ, N_{ldr}, g, level ← ♭, start ← 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 ← SELECT(Ñ)
11       SEND(g, Join⟨s⟩)
12     case Components⟨C_g⟩
13       if s ≠ g then
14         {leader, C} ← SELECT-LEADER(C, C_g, next-level)
15         if parent = USER then
16           parent ← leader
17       else
18         leader ← s
19       foreach t ∈ N ∪ {s} do
20         SEND(t, Done⟨s, leader⟩)
21     case Done⟨t, leader_t⟩
22       if t ≠ s then
23         N_ldr ← UPDATE(N_ldr, N, t, leader_t)
24       Ñ ← Ñ ∖ {t}
25       if Ñ = NIL then
26         level ← ♭
27         SEND(leader, Leader⟨N_ldr ∖ {leader}, C, s = g, next-level⟩)
28     case Leader⟨N', C', single, level'⟩
29       if start then
30         N ← N', C ← C', N_ldr ← NIL
31       else
32         pairs ← {({t,r}, w) : (t, N̂) ∈ C ∧ r ∈ C' ∧ (r, w) ∈ N̂}
33         N ← COMBINE(N, N'), C ← C ∪ C'
34       start ← ¬start ∨ single
35       if start then
36         if N = NIL then
37           leader ← USER
38           foreach t ∈ C do
39             SEND(t, Search)
40         else
41           level ← level', next-level ← level + 1
42           Ñ ← N ∪ {(s, 1.5)}
43           g ← SELECT(Ñ)
44           SEND(g, Join⟨s⟩)
```

is initially empty. Whenever an agent is selected as a leader in later stages of the algorithm, it has to perform similar tasks. For example, when two agents are joined at level 3, each one can have up to 4 components (all of which are single agents). The agent that is selected as the leader must receive the list of its components, which can include up to 7 agents. It must also receive the list of its neighbors at level 4. This list is compiled by the two joining agents of level 3, from the Done messages that they had received from all of their neighbors at level 3.

If the message implies $N = \text{NIL}$, it means that s is the top-level leader responsible for the group of all the agents, and the search phase can start. In this case, s sends Search message to all the agents (including itself), which effectively implements a synchronization barrier between the two phases of groups partitioning and solutions search (lines 36–39).

In the general case, the agent will receive two Leader messages (line 28), and will unite the received neighbors and components (the start variable is used for this task) (lines 29–34). Uniting the neighbors also needs to combine the weights correctly — weights of connections to the same agent are multiplied together. See Algorithm 4.4 for combining weights details.

Let us now consider the join negotiation process. After receiving a Leader message (line 28), s (provided it is not the top-level leader) uses the SELECT() function to pick a neighbor to which it is connected with minimal weight (reverse constraint tightness), and sends it a Join request (lines 41–44). Note that if all neighbors have become unavailable (joined another agent), s will successfully try to join itself, since it is considered to be connected to itself with a unique maximum weight.

A response to the Join message can come in the form of a NoJoin (line 9) or a Join message (line 4). Only the former of these is really a response, as the latter bears semantic meaning of a response, but is sent independently of the Join message originating from s.

In case of a NoJoin, s repeats its attempt (lines 10–11), in the same way as it did at the end of handling the Leader message.

In case of a Join, there are two possibilities. If the message comes from the same agent which s picked (i.e., g), this successfully concludes the join negotiation (line 6). When a join negotiation is completed, both agents need to select a leader from the set of agents in the combined (merged) set. They start by sending each other the list of their own components, using the Components messages. Otherwise, the join attempt originates with an agent that was not selected by s, so it is rejected with NoJoin (line 8), and s continues to wait for a response from its selected g.

When s receives a Components message, it computes the list of components which will be later sent to the leader (line 14). The leader is picked from the united components list using the SELECT-LEADER() procedure (Algorithm 4.2), which picks an available agent for this task, and marks it as taken (puts non-0 level in the appropriate field). A special case is when s unites with itself (because it could not find a node to merge with), in which case it just ascends to the next level, and no

updating takes place (line 18). SELECT-LEADER() is deterministic, and chooses an agent with a maximal number of connections across the merging components lists; a possible implementation is shown in Algorithm 4.2. Done messages are then sent to all the neighbors of s (including s itself), notifying them of the chosen leader, so that they can update the neighbors list of their own leader (lines 19–20).

Algorithm 4.2: 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 $\leftarrow \left|(\{t\} \times N') \cap \left((C \times C') \cup (C' \times C)\right)\right|$
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** $(\text{leader}, \cdot, \text{level}_{\text{ldr}}) \in C$ **then**
10 $\text{level}_{\text{ldr}} \leftarrow$ level

In case of a Done message (line 21), the update process (shown in Algorithm 4.3) is what happens first, and Done messages from the selected partner and s itself are ignored (lines 22–23). UPDATE() also combines the weights of connections to multiple neighbors which are now components of a neighboring leader. Then, the sending agent is removed from the dynamic neighbors list (line 24), and the list is tested for emptiness. Since the list also contains s, this condition will only come true after s sends itself a Done message (which happens after leader selection). When the dynamic neighbors list is finally empty, s resets its level tag to $\mathfrak{p}$ (intended only for Leader and Search messages), and sends the leader an initiation message (lines 25–27).

Algorithm 4.3: UPDATE(N_{ldr}, N, t, leader_t): Updating leader's neighbors list.

Input : leader's neighbors list N_{ldr}, current neighbors list N, a neighbor t, its leader leader_t
Output: updated N_{ldr}

1 **select** $(t, \text{weight}) \in N$ **do**
2 **if** $(\text{leader}_t, \text{old-weight}) \in N_{ldr}$ **then**
3 old-weight $\leftarrow$ old-weight $\cdot$ weight
4 **else**
5 $N_{ldr} \leftarrow N_{ldr} \cup \{(\text{leader}_t, \text{weight})\}$

Algorithm 4.4: COMBINE(N, N'): Combining weights in neighbors lists.

Input : neighbors lists N and N'
Output: updated N

```
1 forall (t, weight) ∈ N' do
2   if (t, old-weight) ∈ N then
3     old-weight ← old-weight · weight
4   else
5     N ← N ∪ {(t, weight)}
```

4.4 Algorithm correctness

An informal proof of the correctness of the partition algorithm (Algorithm 4.1) can be constructed as follows: show that Algorithm 4.1 results in a correct partitioning of agents into a binary tree of groups (as shown in the example in Figure 3.2).

The control flow of the algorithm, in a given agent, can be modeled as shown in Table 4.1. 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 in a finite time; receiving a Join answer from some agent g; exchanging Components messages with g; sending Done messages to all neighbors; receiving Done messages from all neighbors, and sending a Leader message. This establishes termination for Algorithm 4.1.

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.

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 new neighbors list is empty, send Search messages to all agents and go to `Receive`. Otherwise, go to `NoJoin`.
`Search`	Final state.

Table 4.1: Control flow of Algorithm 4.1.

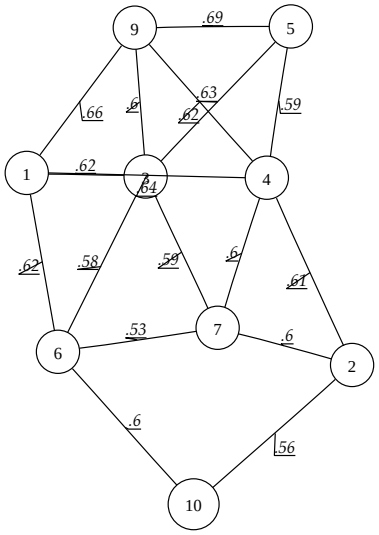
Or, it receives one Leader message from itself with single = TRUE (which happens when an agent did not succeed in joining another agent), and in this case no new node is formed (level is incremented by 1, the components set remains the same, and the neighbors list is updated).

The binary partitioning tree is essentially built bottom-up, resulting in a tree similar to Figure 3.2. This establishes correctness for Algorithm 4.1.

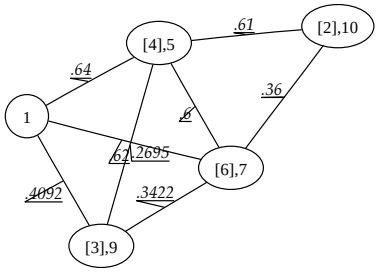
4.5 A partition example

Figures 4.3 and 4.4 show a complete trace of GROUP-PARTITION() execution on a randomly generated CSP, with $p_1 = p_2 = 0.4$.

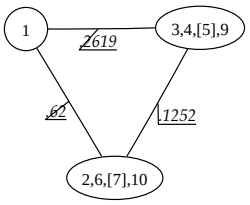
A trace of messages received by the agents during the partitioning process is shown in Appendix A. The order between different agents' messages is the result of synchronization performed by the system's output stream. Note the Search message sent by agent 8 to itself in line 39.



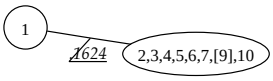
(a) Initial CSP, $p_1 = p_2 = 0.4$. This is a level-0 partition graph, with constraint weights average approximately equal to $1 - p_2$. Note that agent 8 (not shown here) is disconnected from the graph, and thus immediately sends itself a Search message.



(b) Partition graph at level 1. Components are listed in each node, with the representative agent shown in brackets. Note that agent 1 has not succeeded to join another agent. As a result, it ascended to level 1, and is still able to be selected as a leader in the future.



(c) Partition graph at level 2. Here we see that agent 1 will only join with a representative agent in (the last) level 4, since (leader) agents 5 and 7 are connected by a minimal weight.



(d) Partition graph at level 3. Only two agents are left, and they will merge at level 4.

Figure 4.3: Successive levels formation during an execution of GROUP-PARTITION() on a 10 agents random graph.

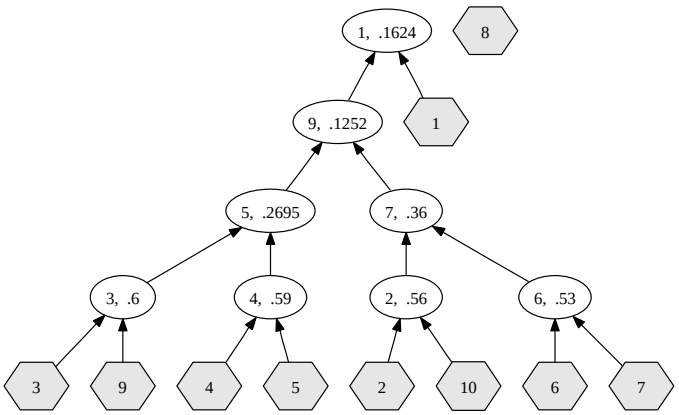


Figure 4.4: Partition tree, showing representative agents and constraint weights between two subgroups of each representative agent. Agents 1 and 8 produce independent partial solutions.

Chapter 5

Concurrent, Hierarchical Search

After the preprocessing, the agents are partitioned into a hierarchy of groups, and each group has one of the agents as its leader. The first search algorithm we will present achieves concurrency by combining partial solutions concurrently. Partial solutions correspond to the groups that were generated by the partition process. Leaders of groups control the generation of partial solutions of the groups they lead, and deliver consistent partial solutions to the leaders of their next higher level. Leaders check the consistency of candidate partial solutions for their group by requesting the involved agents to check constraints.

Since some agents may get their Search message earlier than others, and hence some agents may receive search phase messages before receiving the Search message, a distinct tag `s` is employed for all messages in this phase.

Each agent takes care of two missions. At the primitive level, it can answer queries for checking consistency of constraints that involve it. These queries are asked during the search process, which is performed by group leaders. Case Check, in Algorithm 5.1, executes in each agent s, regardless of whether it is assigned a group, and regardless of its level in the hierarchy of groups.

During the search itself, agents produce solutions and send them up in the groups hierarchy. This begins at level 0 in the hierarchy, when all agents send their domains to their parents, as shown in the Search case of the algorithm. The Answer and Assignment cases for the representative agents, shown in the remainder of Algorithm 5.1 and detailed in Algorithm 5.2, solutions from child group agents are combined, and suitable partial assignments are sent up in the hierarchy.

In order for the solving process to avoid getting stuck in local minima of the search space ("bad" subtrees), the iteration over partial assignments pairs is performed in random order. For this purpose, once a partial assignment arrives from one of the children, its unions with existing partial assignments from the other child are added to the assignments store. Later, when the leader requires an assignment to test, a random assignment is picked, and removed, from the store (lines 6–7 of Algorithm 5.4).

To efficiently facilitate these operations, the store can be implemented using

Algorithm 5.1: DISHS-SOLVE(s): Searching for solutions.

Input : agent s, output from Algorithm 4.1, domain D
Output: a global solution is sent to USER
Locals : row $\leftarrow 0$, Pending$[\cdot] \leftarrow 0$, Solutions$[\cdot]$, Iterator $\leftarrow$ NIL, requests $\leftarrow 1$

```
1  loop forever do
2    switch RECEIVE() do
       ▷ Continuing Algorithm 4.1...
3      case Search
4        level ← 𝔰
5        forall v ∈ D do
6          SEND(parent, Assignment⟨s, {⟨s, v⟩}⟩)
7        SEND(parent, Assignment⟨s, STOP⟩)
8      case Check⟨t, row', {⟨s, v⟩, ⟨r, w⟩}⟩
9        SEND(t, Answer⟨row', CHECK(v, r, w)⟩)
10     case Answer⟨row', ok⟩
11       if Pending[row'] ≠ 0 then
12         if ¬ok then
13           Pending[row'] ← 0
14           requests ← requests + 1
15           PROCESS-REQUEST()
16         else
17           Pending[row'] ← Pending[row'] − 1
18           if Pending[row'] = 0 then
19             SEND(leader, Assignment⟨s, Solutions[row']⟩)
20     case Request
21       requests ← requests + 1
22       PROCESS-REQUEST()
23     case Assignment⟨t, partial⟩
24       ITERATOR-ADD(Iterator, t, partial)
25       PROCESS-REQUEST()
```

Algorithm 5.2: PROCESS-REQUEST(): Process pending requests for assignments to be sent up in the hierarchy.

Input : see Algorithm 5.1
Output: actions necessary to ensure partial solutions flow up the hierarchy are taken

```
1  while requests > 0 do
2    assignment ← ITERATOR-NEXT(Iterator )
3    if assignment = NIL then
4      forall child ∈ ITERATOR-REQUESTS(Iterator ) do
5        SEND(child, Request)
6      return
7    else if assignment = STOP then
8      SEND(leader, Assignment⟨s, STOP⟩)
9      return
10   else
11     forall {r,q} ∈ pairs do
12       select {⟨r,v⟩,⟨q,w⟩} ∈ assignment do
13         SEND(r or q, Check⟨s, row, {⟨r,v⟩,⟨q,w⟩}⟩)
14     Pending[row] ← |pairs|
15     Solutions[row] ← assignment
16     row ← row + 1
17     requests ← requests − 1
```

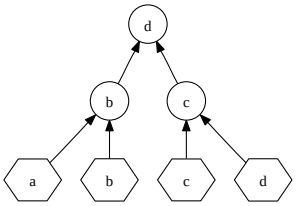


Figure 5.1: Partition hierarchy of the small constraint network.

a complete binary tree, as shown in Appendix B, providing logarithmic times for all operations. In our implementation, however, we use a dynamic array, which is more efficient in practice, providing constant amortized operation times.

5.1 DisHS survey

Let's return to the example in Section 4.1. The partition hierarchy of the graph-coloring problem is reproduced in Figure 5.1.

The solving proceeds as follows. Primitive agents a, c and d send partial assignments $\langle x = 1\rangle$, $\langle x = 2\rangle$, $\langle x = 3\rangle$ (where x is a, c and d, respectively — each agent sends three partial assignments) to the leaders b and c. Agent b sends an additional $\langle b = 4\rangle$ to its leader, which happens to be itself (the leader of primitive

(a) b sends consistent pairs to d.

a	b
1	2
1	3
1	4
2	1
2	3
2	4
3	1
3	2
3	4

(b) c sends consistent pairs to d.

c	d
1	2
1	3
2	1
2	3
3	1
3	2

(c) d produces solutions.

a	b	c	d
1	2	3	1
1	2	3	2
1	3	2	1
1	3	2	3
1	4	2	1
1	4	2	3
...			

Table 5.1: Partial assignments sent by leaders up in the partition hierarchy.

agent b is b). Afterwards, leaders b and c prune inconsistent pairs of these partial assignments, and send the consistent pairs to their leader d.

The partial assignments sent up in the partition hierarchy are described in Table 5.1. Note, however, that the assignments aren't necessary sent in the given order.

While leaders b and c don't need to query other agents in order to prune inconsistent value pairs (since they have full knowledge of the relevant constraints), d prunes inconsistencies using Check queries to a, b and c, and only then produces complete solutions to the DISCSP.

5.2 Algorithm primitives

Algorithm 5.1 takes as its input the output variables of Algorithm 4.1. The algorithm also makes use of several key variables:

- Solutions$[\cdot]$: a table of unions of partial assignments, which await resolution of their validity. That is, not all answers to check requests for these unions were received yet, and those received so far did not reveal constraints conflicts.
- Pending$[\cdot]$: mirrors the Solutions$[\cdot]$ table, but holds numbers of check requests not yet answered.
- row: incrementing row index for the two tables above. This index is also attached to the check requests, and is returned with the corresponding answers.
- Iterator: an iterator over partial assignments unions. The iteration is performed in random order.

- requests: number of assignments which have to be produced. This is essentially a set of tokens which are passed between Algorithm 5.1 and Algorithm 5.2.

The following message types are used during the algorithm's execution:

- Search: initiates the search phase of DISHS. Causes all primitive agents to send their domains to their respective parents, followed by a STOP assignment, indicating that no more assignments are available.
- Check⟨t, row, constraint⟩: message to a primitive agent from representative agent t, requesting to check a specific constraint (including values) belonging to that agent.
- Answer⟨row, result⟩: boolean answer from a primitive agent, with row equal to the corresponding field in the respective Check query.
- Assignment⟨t, partial⟩: an assignment from a representative agent t, sent to the leader of its enclosing group. Each leader receives Assignment messages from its two children.
- Request: a message from a representative agent to one of its children (representative agents of one of the two subgroups), requesting one more assignment.

Since each agent is a primitive agent, and can also be a representative agent for some group, there are essentially two processes competing within each agent in the second phase of DISHS. For this reason, Check and Answer messages are given higher priority than Assignment and Request messages, since not handling the check queries in time can delay other representative agents. Therefore, while there are Checks or Answers in the incoming messages queue, Assignment and Request messages are waiting.

5.3 Algorithm analysis

At the end of the partition phase, described in Chapter 4, all agents receive a Search message (tagged with level $\mathfrak{p}$). Then, each agent iteratively sends its domain to its *parent*, using the Assignment messages, followed by a STOP Assignment message. The STOP type of message is ultimately used to discover that there is no solution to the DISCSP. At the same time, the level is also changed to $\mathfrak{s}$, which is the distinctive level for the solving phase of DISHS.

The solution search process itself begins once representative agents of level-1 start receiving the singleton partial assignments from their child primitive agents. Each leader adds received Assignments to its unified assignments Iterator, and attempts to produce the required number of assignments, as indicated by the requests variable, initially 1.

During this attempt to produce united assignment, shown in Algorithm 5.2, the representative agent may forward the request to its child agents using the Request message. This happens when no more partial assignments unions can be produced by Iterator.

When the iterator succeeds in producing another combined assignment, the representative agent sends one Check message for each cross-subgroup constraint. The message carries values extracted from that assignment, to an agent which can answer whether the constraint is satisfied. Consequently, the Pending and the Solutions tables are updated with the information about a possible partial solution, and the requests counter is decremented.

Once the representative agent receives a negative Answer message in response to a previously sent Check message, it increments the requests counter back. Next, it attempts to produce another partial solution up in the hierarchy, since the unified assignment for which the Answer message arrived is not a valid partial solution. If, however, all Answer messages were positive, the partial solution is valid, and is sent up in the hierarchy of the partition tree.

This process continues until USER, a destination representing the system encapsulating the DISHS algorithm, receives an Assignment message with either a solution to the DISCSP, or a STOP, indicating the absence of a solution.[1]

5.4 Algorithm optimization

5.4.1 On-demand partial solutions

In general, Algorithm 5.1 can be written in a much simpler manner than shown here. All representative agents may write information about combined partial assignments to the Solutions and the Pending tables, removing the need for assignment requests sent down in the hierarchy, and greatly simplifying the concept of Iterator.

However, this presents a problem of explosive growth in the number of messages not yet handled at any moment. The underlying problem is that most solvable DISCSP problems, even randomly generated, contain search space regions abundant with partial solutions. Hence, once a group of agents, led by a representative agent, encounters such a region, it will send a massive amount of Assignment messages with partial solutions up in the partition hierarchy. Moreover, it will also send a massive amount of Check messages to verify these partial solutions, and thus overflow the agents performing the checks, causing them to indefinitely postpone check requests from other representative agents.

Thus, a single representative agent stumbling upon a partial solutions-abundant search subspace may cause constraint checks starvation to other representative

[1] We assume here that the DISCSP constraints graph is connected. If it has two or more components (which may happen, e.g., when generating random problems for an experiment). USER will receive at least one Assignment message from the top-level agent of each component.

agents, which can delay the production of their respective partial assignments. This happens in addition to the general overflow of the message queues in the agents network. The starvation effect is dominant in causing the system to fail to produce any solutions to the DISCSP in a reasonable time.

Consequently, the second phase of DISHS, as shown here, uses the *on-demand* concept of producing partial solutions: send just one consistent assignment up in the hierarchy, and respond to additional requests.

One might argue that it would be beneficial to attempt to produce some $k > 1$ partial solutions each time. However, the round-trip delay of a partial solution request does not matter in practice, since in general a single partial solution received from a child agent allows the representative agent to try many combined partial assignments (equal to the number of partial solutions already received from the other child agent).

5.4.2 Cached answers

Constraint check requests are sent with two agent-value pairs. There is no need to perform the same check over and over, since the results can be stored as *cached answers* in the Iterator. Whenever a partial assignments union is requested using ITERATOR-NEXT(), each constraint in the assignment being combined (i.e., two agents and their respective values) is resolved using the cached answers table.[2]

Cached answers are a double-edged sword, since the checks against the nogoods table are consecutive, while the constraint checks requested using Check messages are performed in parallel. On the other hand, analysis of appropriately ordered consecutive constraint checks in Section 3.2 suggests that they are beneficial to the overall performance of DISHS, and this is indeed what happens in practice.

With the consecutive checks against the cache, it is only natural to extend this mechanism to the representative agent itself, which checks constraints that are owned by itself, before sending Check messages for remote constraints. This also improves performance of the search phase of DISHS.

Algorithm 5.3 shows the process of addition of a partial solution to the representative agent's assignments iterator. We see that the partial solution is added to one of two queues, each one holding solutions received from one of two child agents (the STOP solutions terminator is handled as a special case). After that, unions of the given partial solution with all the solutions from the other queue are added to the united assignments list, from which they are later picked in random order. Finally, since the child agents delivered a partial solution, its requested indicator is reset.

Algorithm 5.4 shows the process of iteration over united pairs of partial solutions, which have been received from the child agents. First, if the list of unified assignments is empty, ITERATOR-NEXT() returns NIL, unless both child agents sent

[2]Each separate check against the table is, of course, counted as a constraint check in the experiments.

Algorithm 5.3: ITERATOR-ADD(Iterator, t, partial): Add a partial solution to the assignments iterator.

Input : Iterator, source t, partial solution partial
Output: partial is registered in the iterator
Static : $\text{sources}_{0,1} \leftarrow$ UNASSIGNED, $\text{queues}_{0,1} \leftarrow$ NIL, $\text{closed}_{0,1} \leftarrow$ FALSE, $\text{requested}_{0,1} \leftarrow$ FALSE, assignments $\leftarrow$ NIL, Cache$[\cdot]$

```
1  i ← 0
2  if sources_0 ∉ {UNASSIGNED, t} then
3      i ← 1
4  sources_i ← t
5  if partial = STOP then
6      closed_i ← TRUE
7  else
8      queues_i ← queues_i ∪ {partial}
9      assignments ← assignments ∪ {partial ∪ partial' : partial' ∈ queues_{1−i}}
10     requested_i ← FALSE
```

STOP terminators, in which case it also returns STOP (since no more assignments can be produced). Since the unions are constructed in Algorithm 5.3, ITERATOR-NEXT() randomly picks and removes an assignment from the existing list. As was mentioned before, an approach outlined in Algorithms B.1, B.2, and B.3 can be used. However, our implementation of DISHS uses a dynamic array to implement the unified assignments list, which implies constant amortized cost for addition and removal of elements.

Finally, ITERATOR-NEXT() goes over the constraints list and checks whether they are recorded in the cached answers table. If a constraint is recorded as a conflict, the algorithm returns FAIL (taking care of FAILs is not shown in Algorithm 5.2 for the sake of clarity). If no cached conflicts have been encountered, ITERATOR-NEXT() returns the unified assignment, together with the list of non-cached constraints which should be checked. Usage of this list of constraints is also implicit in PROCESS-REQUEST().

Algorithm 5.5 shows what happens when ITERATOR-NEXT() reports that no more unified assignments are available, and PROCESS-REQUEST() (Algorithm 5.2) calls ITERATOR-REQUESTS() in order to find out to which child agents it will send Request messages. The algorithm is quite simple — it does not include a child agent in the returned set if it produced no partial solutions since the previous call to ITERATOR-REQUESTS(), if it sent a STOP terminator (indicating that no more solutions are available), or if it is a primitive agent. Sending Request messages to primitive agents is meaningless, since such messages are handled by representative agents; primitive agents send their whole domain to their respective parents in the beginning of the search phase (Algorithm 5.1).

Algorithm 5.4: ITERATOR-NEXT(Iterator, pairs): Produce next unified assignment from the iterator.

Input : Iterator, constraints list pairs
Output: a unified assignment is returned, together with the list of constraints to check
Static : see Algorithm 5.3

```
1  if assignments = NIL then
2     if closed_0 ∧ closed_1 then
3        return STOP
4     else
5        return NIL
6  candidate ← random element from assignments
7  assignments ← assignments ∖ {candidate}
8  checks ← NIL
9  forall {r, q} ∈ pairs do
10    select {⟨r, v⟩, ⟨q, w⟩} ∈ candidate do
11       constraint ← {⟨r, v⟩, ⟨q, w⟩}
12       if constraint ∈ Cache then
13          if ¬Cache[constraint] then
14             return FAIL
15       else
16          checks ← checks ∪ {constraint}
17 return ⟨candidate, checks⟩
```

Algorithm 5.5: ITERATOR-REQUESTS(Iterator): Determine the child agents which need to be sent requests for additional assignments.

Input : Iterator
Output: a set of 0–2 children is returned
Static : see Algorithm 5.3

```
1  requests ← NIL
2  forall i ∈ {0, 1} do
3     if ¬(requested_i ∨ closed_i ∨ sources_i = UNASSIGNED ∨ |queues_i[0]| = 1) then
4        requests ← requests ∪ {sources_i}
5        requested_i ← TRUE
```

5.5 Algorithm correctness

We now present an informal proof of the correctness of the second phase of the DISHS algorithm. We show that Algorithm 5.1 indeed results in all correct solutions sent to USER, followed by an indicator that the search space has been exhausted.[3]

Let us follow the control flow of the algorithm. Upon receiving a Search message, all level-1 representative agents receive a singleton Assignment message for each value in the domains of their child primitive agents. Moreover, each representative agent is assigned, via the requests variable, a requirement to produce one consistent partial solution to be sent up in the hierarchy.

We note that it is not possible for the requests variable to decrease to 0, and then increase back to 1, without an attempt to produce a partial solution. Such an attempt is performed via PROCESS-REQUEST() subroutine, shown in Algorithm 5.2.

In Algorithm 5.2 we see that, unless Iterator is exhausted, requests is reduced immediately after sending Check messages to the agents possessing knowledge about the constraints which need to be checked (lines 11–17). In essence, the reduction in requests is passed as a token with the constraints check requests. When the respective Answer messages are received (line 10 of Algorithm 5.1), this token will either return to requests, if one of the constraint checks failed (lines 12–15), or will be transformed into a partial solution, which will be sent to the leader of the current representative agent (lines 16–19).

Also, if Iterator is exhausted, an additional Request is sent to one or both of the child agents, causing them to produce more partial solutions. Once these solutions are received in the representative agent, Algorithm 5.2 will execute again, with a refilled Iterator. Or, the children's search space is exhausted, in which case the corresponding STOP indicator propagates up in the hierarchy.

Ultimately, each representative agent either produced a partial solution and sent it up in the hierarchy, is attempting to produce a partial solution, or requested its children to produce one, so it can try to produce a partial solution (or all of these together). At no time can the algorithm enter a deadlock, and no partial assignment is tried more than once. This establishes termination for Algorithm 5.1.

Correctness of the algorithm follows from the way in which the representative agents assemble and check their partial assignments from the partial solutions received from their child agents. A united partial assignment will be authorized as a valid partial solution only when all the constraints across the partial solutions of the two sub-components are confirmed not to be in conflict by the agents which possess knowledge about these constraints.

By induction, starting from the apriori valid singleton partial assignments sent by the primitive agents, each united assignment with non-conflicting constraints across the two sub-components is a valid partial solution for the agents component

[3] Since Algorithm 5.1 presents an on-demand version of the search for solutions, just one DISCSP solution will be sent to USER. However, the encapsulating system may request additional solutions via Request messages to the top-level representative agent.

lead by a given representative agent. Thus, partial solutions sent to USER by the top-level agent are valid solutions to the DISCSP. This establishes correctness for Algorithm 5.1.

5.6 Experimental evaluation

The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance — time, in the form of steps of computation [14, 25], and communication load, in the form of the total number of messages sent [14].

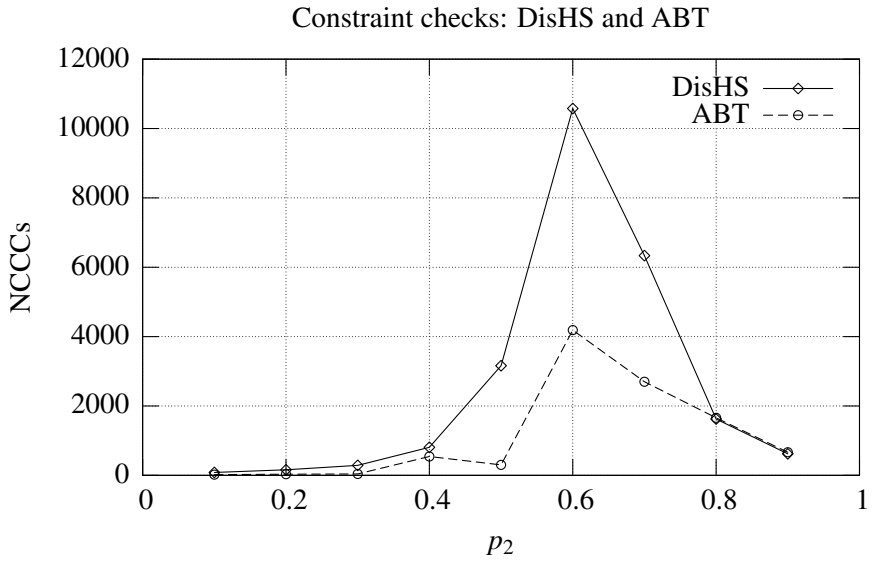
Non-concurrent steps of computation are counted by a method similar to the clocks synchronization algorithm of Lamport [13]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent's counter. When an agent receives a message, it stores the data received together with the corresponding counter. When the agent first uses the received counter it updates its counter to the largest value between its own counter and the stored counter value which was carried by the message [16]. By reporting the cost of the search as the largest counter held by some agent at the end of the search, a measure of non-concurrent search effort that is close to Lamport's logical time is achieved [13]. If instead of steps of computation, the number of non-concurrent constraints check is counted (NCCCs), then the local computational effort of agents in each step is measured [16].

Experimental evaluation of the DISHS 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.

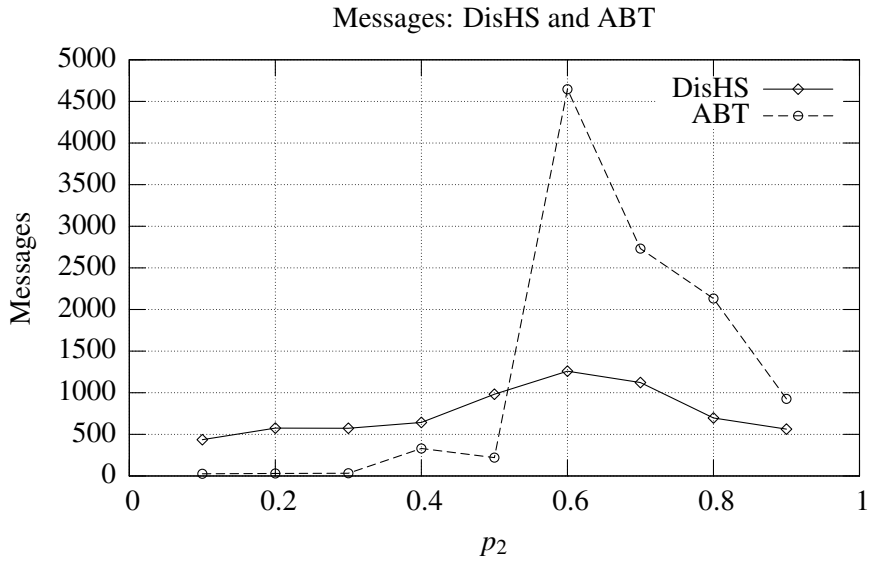
Experiments were conducted on random networks of constraints. The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables (*constraint density*) and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables (*constraint tightness*) [17, 19].

In Figure 5.2, we see a comparison of DISHS and ABT on random problems of moderate complexity. ABT outperforms DISHS in the number of non-concurrent constraint checks performed by a ratio of about 2.5 in the phase transition region (Figure 5.2(a)). DISHS uses much less messages than ABT in the same region (Figure 5.2(b)).

On problems of higher complexity (more agents, larger domains — results not shown here), DISHS's performance drops dramatically. This is probably due to DISHS wasting most of the computation resources on futile attempts to combine partial solutions. One can speculate that some form of backtracking is necessary for good performance of DISCSP algorithms. Chapter 6 describes a different second phase algorithm, which purports to alleviate this deficiency of DISHS.



(a) Non-concurrent constraint checks.



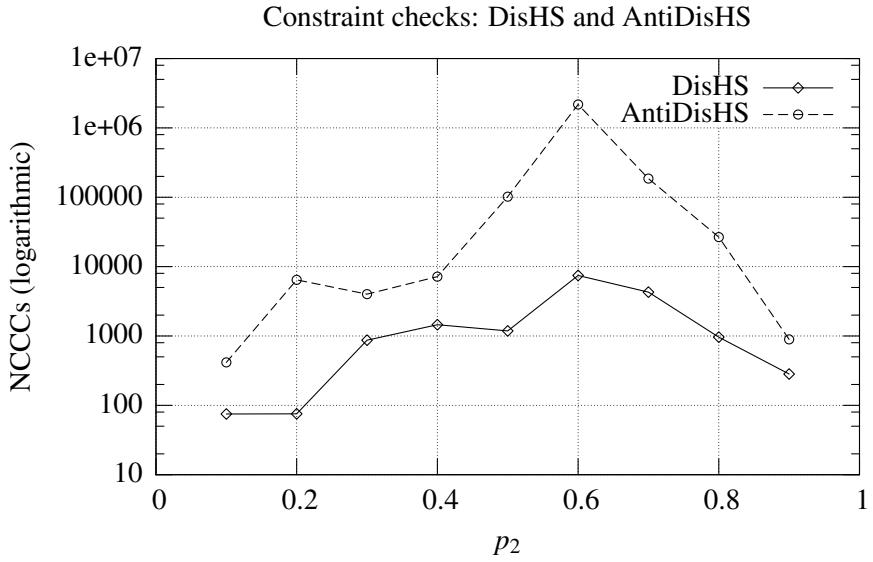
(b) Total number of messages.

Figure 5.2: Comparing DISHS with ABT on random problems with 10 agents, domain size of 10, and $p_1 = 0.5$.

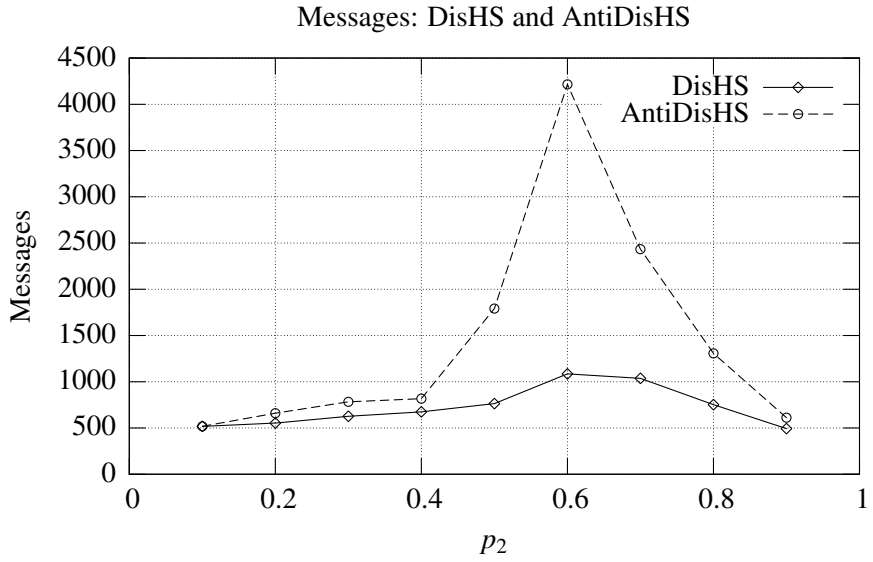
In Figure 5.3, we see a comparison of DISHS against ANTI-DISHS — algorithm similar to DISHS, but with the weights ordering reversed during the groups partitioning phase. That is, instead of pushing hard constraints down in the hierarchy, they are pushed up during the execution of GROUP-PARTITION().

We see that the difference, especially in terms of constraint checks, is tremendous — good partitioning hierarchy is vital to the performance of DISHS.

It is interesting to note that the messages performance of ANTI-DISHS in Figure 5.3(b) is similar to that of ABT in Figure 5.2(b).



(a) Non-concurrent constraint checks. Note the logarithmic scale.



(b) Total number of messages.

Figure 5.3: Comparing DISHS with ANTI-DISHS on random problems with 10 agents, domain size of 10, and $p_1 = 0.5$.

Chapter 6

Descending Requirements Search

The solving phase of hierarchical search is straightforward: each representative agent receives independent partial solutions from its child agents and combines them into a larger partial solutions. Concurrency is achieved by the fact that leader agents at all levels check consistency by sending Check queries to the relevant agents for the constraints being checked. However, the experimental evaluation of Hierarchical Search (DISHS) in Chapter 5 seems to point to the fact that for randomly generated DISCSPs it performs on the average more NCCCs than Asynchronous Backtracking (ABT). One way of improving the concurrent performance of hierarchical search is to try and combine *compatible* partial solutions. This is of course impossible to guarantee, in face of the fact that DISCSPs are NP-complete problems. Still, an attempt to improve the design of hierarchical search is in order.

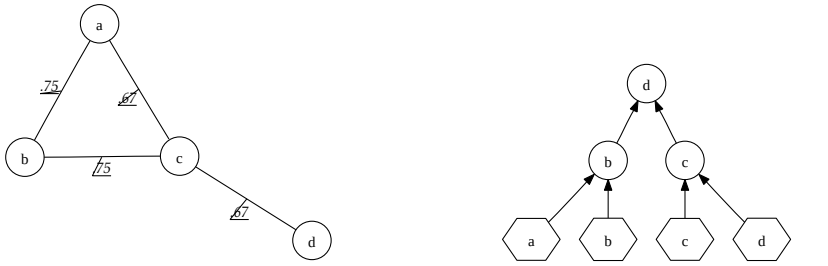
An alternative search phase is presented below, where agents produce assignments which are already compatible with partial assignments sent by their peers.

6.1 General description

In *Descending Requirements Search* (DESRS), presented in Algorithm 6.1, all primitive agents independently initiate empty partial assignments (PAs), which flow up and down in the hierarchy tree formed during the group-partitioning phase of the algorithm (Algorithm 4.1). 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.



(a) Constraint network. (b) The resulting partition hierarchy.

Figure 6.1: A small constraint network and the resulting partition.

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 explanation is designated as the "culprit". The united explanation (excluding the "culprit") is then sent to this agent in a Nogood message [5, 9, 33].

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 [33].

6.2 DesRS survey

Let's return to the example in Section 4.1. The constraint network and the resulting partition hierarchy of the graph-coloring problem are reproduced in Figure 6.1.

Each agent now initiates a backtracking process. We will explore the process originating at leaf a. The flow of the growing partial assignment can be summarized as follows.

Leaf a sends an initial assignment $\langle a = 1 \rangle$ to its leader b. Leader b forwards this partial assignment to leaf b, which then grows it (consistently), and sends $\langle a = 1, b = 2 \rangle$ up in the partition hierarchy. Leader b now forwards the assignment to leaf d via leaders d and c (the latter arbitrarily chooses to forward the assignment to leaf d and not to leaf c). Leaf d then grows the assignment, and forwards $\langle a = 1, b = 2, d = 1 \rangle$ to leader c, which further forwards it to leaf c.

Leaf c grows the assignment, and sends a complete solution $\langle a = 1, b = 2, d = 1, c = 3 \rangle$ to leader c, which forwards it to the user via leader d.

Note that no backtracking happened here. Backtracking details are described in Section 6.4.

6.3 Algorithm primitives

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

- Assignment⟨t, id, PA, primitive⟩: a partial solution which is sent by agent t up in the partition hierarchy. The 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⟨id, exp⟩: a resolved nogood sent to the "culprit" agent. The id field is equal to the id in the Assignment messages with the corresponding (inconsistent) PA, and exp is the resolved explanation, which in DESRS is a set of agents.

Algorithm 4.1 has been implicitly modified to produce the following additional output for each representative agent:

- c_0, c_1: child agents.

- $prim_0, prim_1$: whether the corresponding child agent is a primitive agent (a leaf in the hierarchy).

On the other hand, GROUP-PARTITION() need not produce pairs (the list of constraints between the two sub-components).

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

6.4 DESRS algorithm in detail

At the end of the partition phase, described in Chapter 4, all agents receive a Search message (tagged with level $\mathfrak{p}$). Then, each agent bootstraps its independent search process by sending itself a uniquely-identified Assignment message with an empty partial assignment.

At the same time, the level is also changed to $\mathfrak{s}$, which is the distinctive level for the solving phase of DESRS.

The search process starts when the primitive agents receive their corresponding empty partial assignments. Each primitive agent s initializes the $\{s \rightarrow \langle \text{NIL}, D, \emptyset \rangle\}$

Algorithm 6.1: DESRS-SOLVE(s): Search for a solution.

Input : agent s, output from Algorithm 4.1, domain D, child agents $\mathsf{c}_{0,1}$, primitive child indicators $\mathsf{prim}_{0,1}$
Output: a global solution is sent to USER
Locals : Id-Map[·]

```
1  loop forever do
2    switch RECEIVE() do
       ▷ Continuing Algorithm 4.1...
3      case Search
4        level ← 𝔰
5        SEND(s, Assignment⟨s, s, NIL, TRUE⟩)
6      case Assignment⟨t, id, PA, primitive⟩
7        if primitive then
8          Id-Map[id] ← ⟨PA, D, ∅⟩
9        else if ∃i : t = c_i then
10         if c_{1−i} ∈ PA then
11           SEND(leader, Assignment⟨s, id, PA, FALSE⟩)
12         else
13           SEND(c_{1−i}, Assignment⟨s, id, PA, prim_{1−i}⟩)
14       else
15         i ← RANDOM({0,1})
16         SEND(c_i, Assignment⟨s, id, PA, prim_i⟩)
17     case Nogood⟨id, exp⟩
18       ⟨·, ·, united-exp⟩ ← Id-Map[id]
19       united-exp ← united-exp ∪ exp
20     case Assignment⟨·, id, ·, TRUE⟩ ∨ Nogood⟨id, ·⟩
21       ⟨PA, Values, exp⟩ ← Id-Map[id]
22       v ← NIL
23       while v = NIL ∧ Values ≠ ∅ do
24         v ← RANDOM(Values)
25         Values ← Values ∖ {v}
26         for (r = w) ∈ PA (left-to-right, neighbors of s only) do
27           if CHECK(v, r, w) then
28             exp ← exp ∪ {r}
29             v ← NIL
30             break
31       if v ≠ NIL then
32         SEND(parent, Assignment⟨s, id, ⟨PA, (s = v)⟩, FALSE⟩)
33       else if exp ≠ ∅ then
34         for r ∈ PA (right-to-left) do
35           if r ∈ exp then
36             SEND(r, Nogood⟨id, exp ∖ {r}⟩)
37             break
38       else
39         SEND(USER, Nogood⟨id, exp⟩)
```

mapping, where D is the value domain of s. The agent then grows the empty partial assignment with a value from its domain, and sends the resulting Assignment message to its parent.

Section 6.1 describes the flow of Assignment messages in the system. On their way to a global solution, these messages pass through all primitive agents, which reset the respective id-mappings, and try to grow the PAs by a value from their domain. The initial partitioning aids in this process, since most of the constraints are "pushed down" in the hierarchy, providing DESRS with early opportunities to backtrack using Nogood messages. In other words, the partition of the constraints network generates a distributed version of the fail-first heuristic.

The combined case in line 20 of Algorithm 6.1 describes what happens after the initial processing of "primitive" Assignment and Nogood messages.

First, a value from the set of domain values corresponding to the given id is picked, and removed from the set. If the value is in conflict with an agent in the partial assignment, the chronologically earliest conflicting agent in the PA is added to the growing explanation, and another domain value is attempted.

If a value has been picked, an Assignment message with the new value attached to the partial assignment is sent up in the hierarchy. However, if the domain has been emptied, nogood resolution is performed. In this resolution process, the most recent agent in the corresponding explanation is designated as the culprit, and a Nogood message is sent to this agent, with the resolved explanation. It is also possible that the explanation is empty, in which case the current agent is the agent which initialized the search process for the given id, and the constraints network has no solution.

The reason for recording the chronologically earliest conflicting agent in the PA in the explanation for removing a domain value is that "fixing" a value in chronologically later conflicting agent by backtracking to it will not help — the earlier agent will still conflict with the removed value in the current agent. On the other hand, backtracking with a Nogood message to an agent which is not chronologically latest in the resolved explanation (when the value domain is emptied) would make the search incomplete — the algorithm could miss possibly consistent partial assignments [2].

In a network with n agents, USER will receive n (possibly equal) solutions. If there are no solutions, USER will receive n empty nogoods.

6.5 Algorithm correctness

We now present an informal proof of the correctness of DESRS. We show that Algorithm 6.1 indeed results in a correct solution (or an empty nogood when there is no solution) sent to USER. We do not concern ourselves with iteration over all solutions here — an extension of the algorithm that uses additional USER-originated Nogood messages is quite straightforward.

Upon receiving the Search message (line 3), each agent sends itself an As-

signment message with a unique id (lines 4–5). From now on, processes involving different ids are completely independent, since response messages are always sent with the same id field as the received messages, and the only state is encapsulated in the Id-Map mapping.

The Assignment messages received by representative agents (line 6) are routed in such a way that no cycle is ever formed in the growing partial assignments: a message from a child agent is sent either up in the hierarchy (lines 10–11), or to the other child (lines 12–13), where it is consistently forwarded down to a leaf (lines 14–16). As well, when an Assignment message is sent up in the hierarchy, its PA field covers the whole component for which the sending representative agent is responsible (this can be shown by induction on the number of agents covered by the partial assignment).

Thus, each independent search process corresponds to a possible back-jumping search in a centralized CSP with the same variables — see the discussion on no-goods resolution in Section 6.4.

6.6 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.

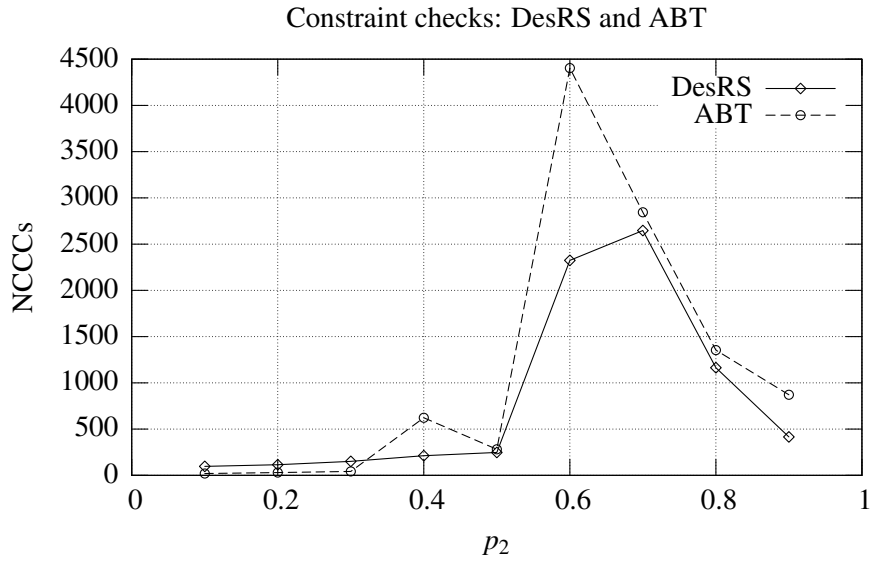
Experiments were conducted on random networks of constraints. The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables (*constraint density*) and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables (*constraint tightness*) [17, 19].

The algorithm has been implemented in the same framework in which the other algorithms, such as ABT, are already implemented, and thus confidence in an adequate comparative evaluation can be established.

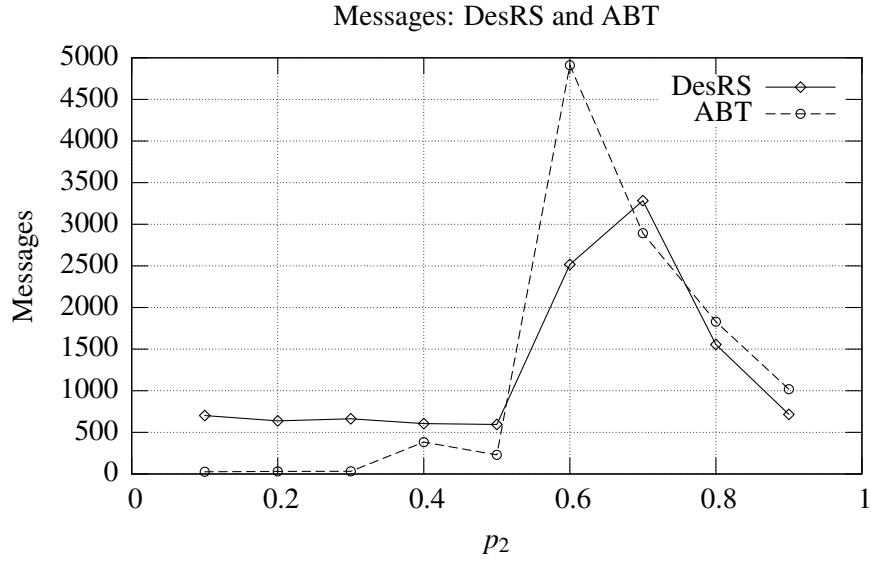
Figure 6.2 compares DESRS to ABT on a set of randomly generated problems of moderate complexity ($n = 10$, $|D| = 10$, $p_1 = 0.5$). Figure 6.3 compares the same algorithms on a set of hard random problems ($n = 20$, $|D| = 10$, $p_1 = 0.4$).

We see that DESRS outperforms ABT on all problems. This is true with respect to both measures of performance — non-concurrent constraint checks [16] 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 6.3). 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 pro-

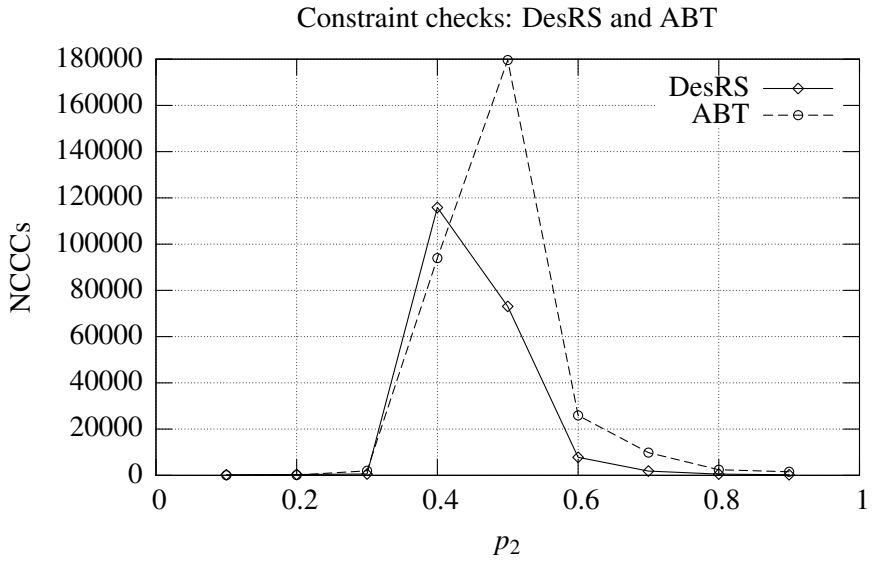


(a) Non-concurrent constraint checks.

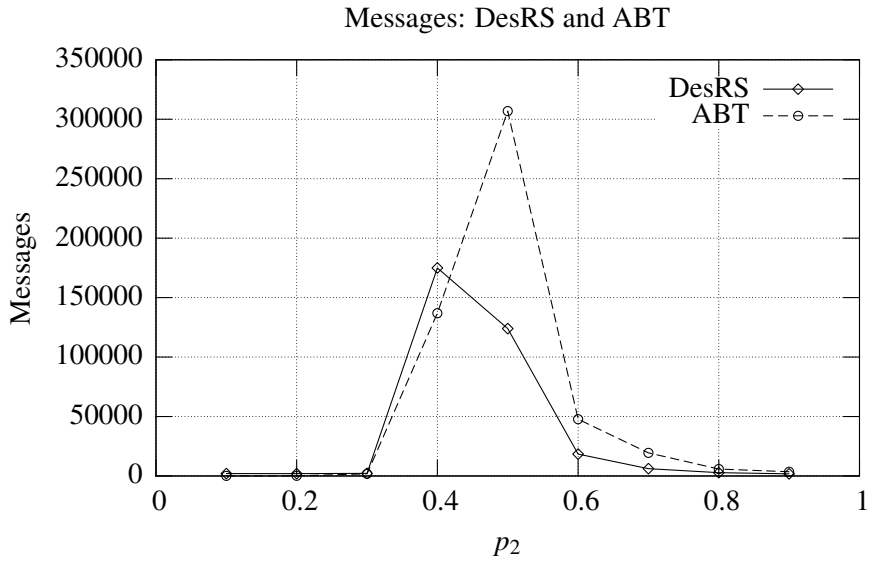


(b) Total number of messages.

Figure 6.2: Comparing DESRS with ABT on random problems with 10 agents, domain size of 10, and $p_1 = 0.5$.



(a) Non-concurrent constraint checks.



(b) Total number of messages.

Figure 6.3: Comparing DESRS with ABT on random problems with 20 agents, domain size of 10, and $p_1 = 0.4$.

cesses. CONCDB [33] generates search processes dynamically during search. Figures 6.4, 6.5 present 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 6.2(a)) [16]. 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 6.3(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 6.6 presents a comparison of DESRS against ANTI-DESRS — algorithm similar to 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.[1]

6.7 Comparison with DISHS

As Section 6.6 shows, DESRS performs much better than DISHS. The difference is quite large, DISHS cannot even handle the hard problem set in which DESRS outperforms ABT (i.e., 20 agents, Figure 6.3).

What are the reasons for this difference? DISHS operates on the premise that concurrent production of independent partial assignments can be beneficial to the distributed search process, if the agents network is optimally partitioned — even if the cost of matching independent PAs can still be high.

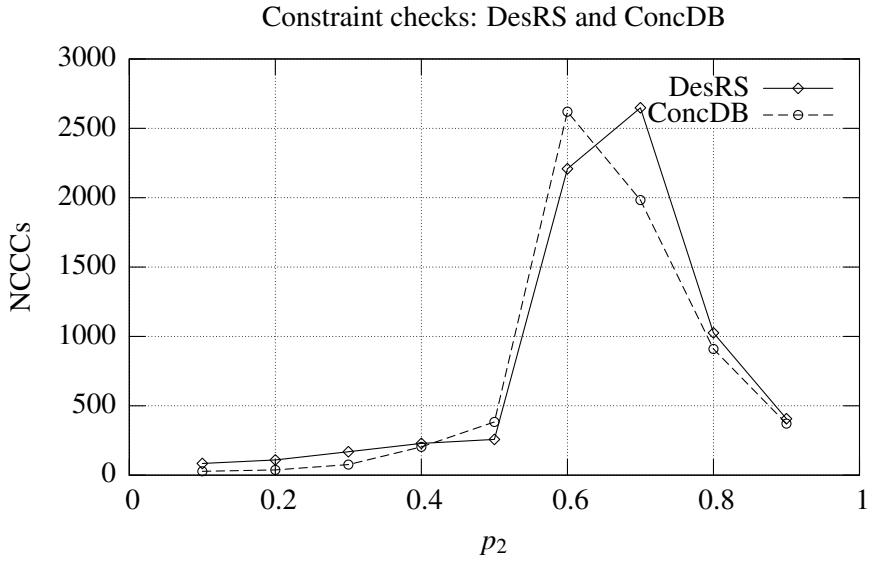
However, DESRS seems to benefit from the concurrency of multiple search processes without sacrificing the incremental construction of partial assignments. Moreover, the randomization of the growing process of PAs after backtracking brings stochastic properties to the algorithm (while still keeping it complete). Such properties are known to improve performance of certain DISCSP topologies [30].

The comparative performance of DESRS, DISHS and ABT suggests that, while exploiting concurrency of distributed search is important, incremental growth of consistent assignments to the DISCSP remains paramount to the efficiency of search process. This is also apparent for other concurrent search algorithms like CONCDB [33], where multiple solutions are constructed incrementally.

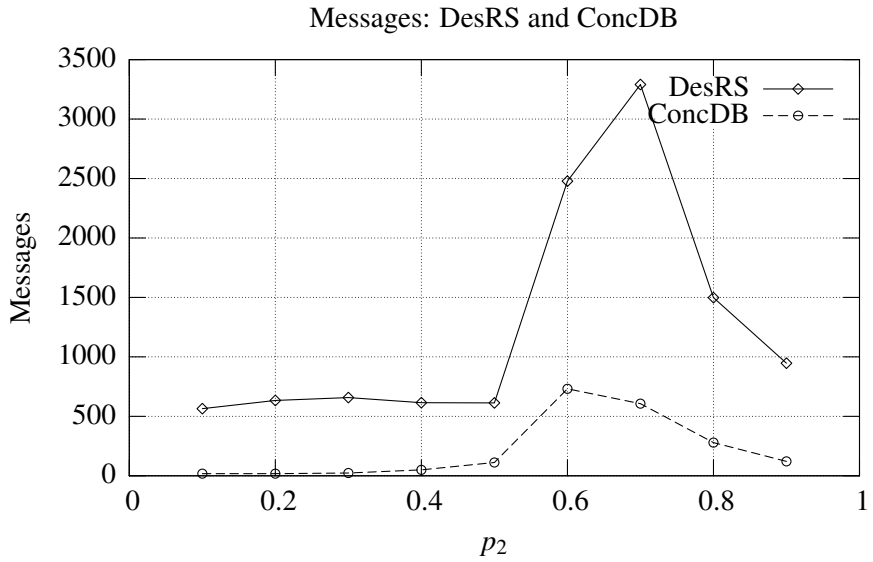
6.8 Discussion

A new search algorithm for distributed constraint problems (DISCSPs) is presented. The new algorithm uses a hierarchy of groups of agents to partition the

[1] The results for DESRS differ from those in Figures 6.3 and 6.5 due to a different experimental environment.

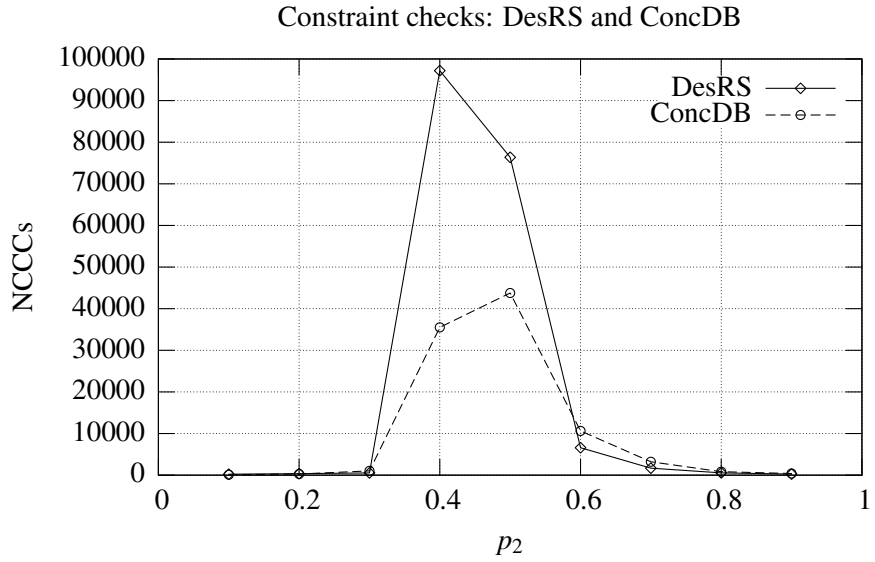


(a) Non-concurrent constraint checks.

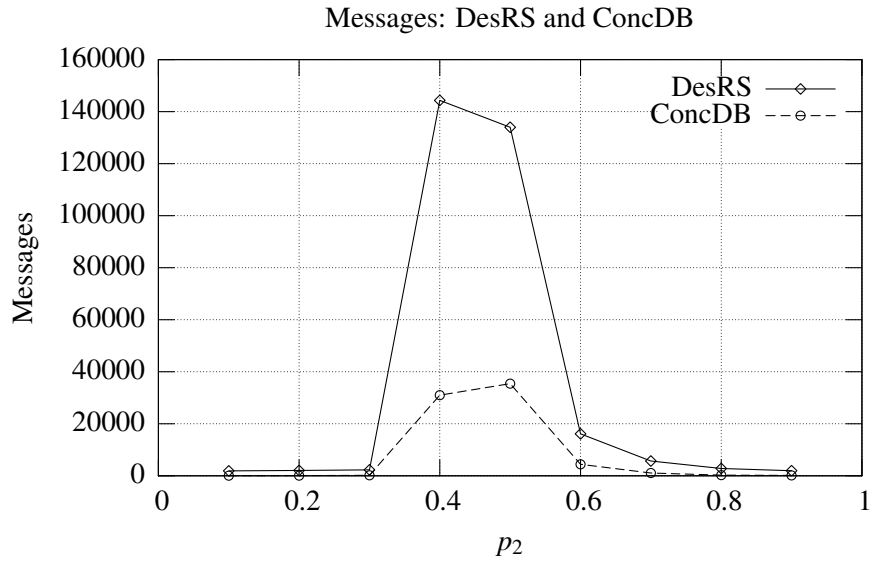


(b) Total number of messages.

Figure 6.4: Comparing DESRS with CONCDB on random problems with 10 agents, domain size of 10, and $p_1 = 0.5$.

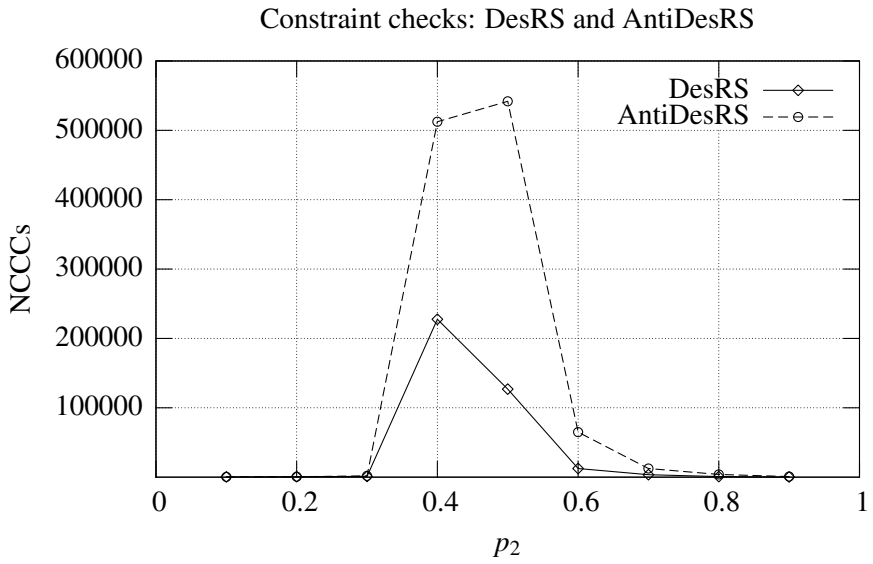


(a) Non-concurrent constraint checks.

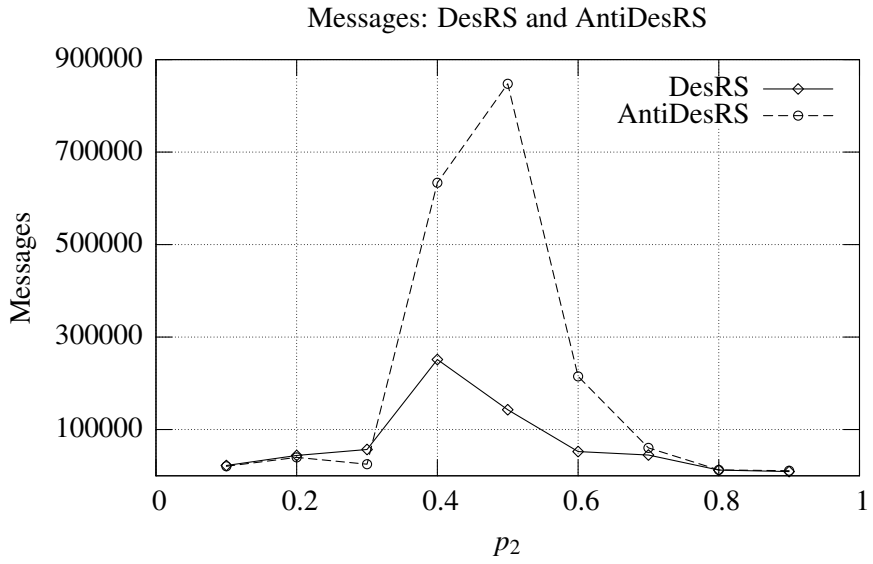


(b) Total number of messages.

Figure 6.5: Comparing DESRS with CONCDB on random problems with 20 agents, domain size of 10, and $p_1 = 0.4$.



(a) Non-concurrent constraint checks.



(b) Total number of messages.

Figure 6.6: Comparing DESRS with ANTI-DESRS on random problems with 20 agents, domain size of 10, and $p_1 = 0.4$.

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.

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 deadends discovered by other search processes [33]. 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.

Chapter 7

Other Applications of Partitioning

The partitioning phase has been used in order to organize agents performing a DISCSP search. The resulting hierarchical grouping was aimed at improving the concurrency of constructing a global solution for the DISCSP. In this chapter we explore other possibilities for applications of the group partitioning algorithm. We consider group partitioning as detached from the search process, and view it separately.

Both search algorithms proposed in this thesis, DISHS and DESRS, are composed of two phases. In the first phase, the agents form a hierarchy of enclosing groups, with representatives selected for each group. The second phase is the search itself, pruning inconsistent solutions, while taking advantage of the hierarchy built during the partitioning phase.

These phases are actually independent. Chapter 4 describes the input and output of Algorithm 4.1. The input to the distributed algorithm is specified by:

- The topology is of a connected undirected graph.
- Each agent has a list of its immediate neighbors.
- Each connection (edge) between agents is assigned a weight.
- A method of combining weights of several edges between groups of agents.
- Unrestricted communication — each agent can send a message to any other agent.

The output after termination of GROUP-PARTITION(), indicated by a Search message received by all agents, is:

- A binary hierarchy of groups is established.
- Each group is either a singleton, or is composed of two subgroups.

- Some of the agents are designated as representative agents — leaders of a group.
- Each agent, composing a singleton group, knows its *parent* — the representative agent of the enclosing group.
- Each representative agent knows its *leader* — the representative agent of the enclosing group. The representative agent also knows its children, and whether they are also representatives, or leafs.
- Each representative agent holds the list of edges between its two subgroups.
- Each representative agent holds the list of agents of its groups.
- The resulting hierarchy has a bias towards connections (primitive or aggregate, combining several primitive edges) with high weight remaining at the low levels of the hierarchy.

The group partitioning algorithm easily accommodates large graphs. Figure 7.1 shows the result of running the algorithm on a 100-nodes network of agents.

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 [3]. 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 4.1:

- The topology of connections among agents (people) connection topology is that of a connected undirected graph. It is reasonable to assume that two people will view the connection between them (or absence of such) as having the same weight. Also, it is probably not interesting to analyze unconnected parts of a social network together, and the network can be assumed to be connected, without loss of generality.
- Each agent has a list of its immediate neighbors: each person knows with whom she regularly converses by phone.
- Each connection (edge) between agents is assigned a weight. In our application, the weight can be the daily proportion of the time during which two given persons talk on the phone.
- A method of combining weights of several edges between groups of agents: weights multiplication seems to suit this purpose well.
- Unrestricted communication — each agent can send a message to any other agent. This is possible if, for example, agent code is executed by cell phone software.

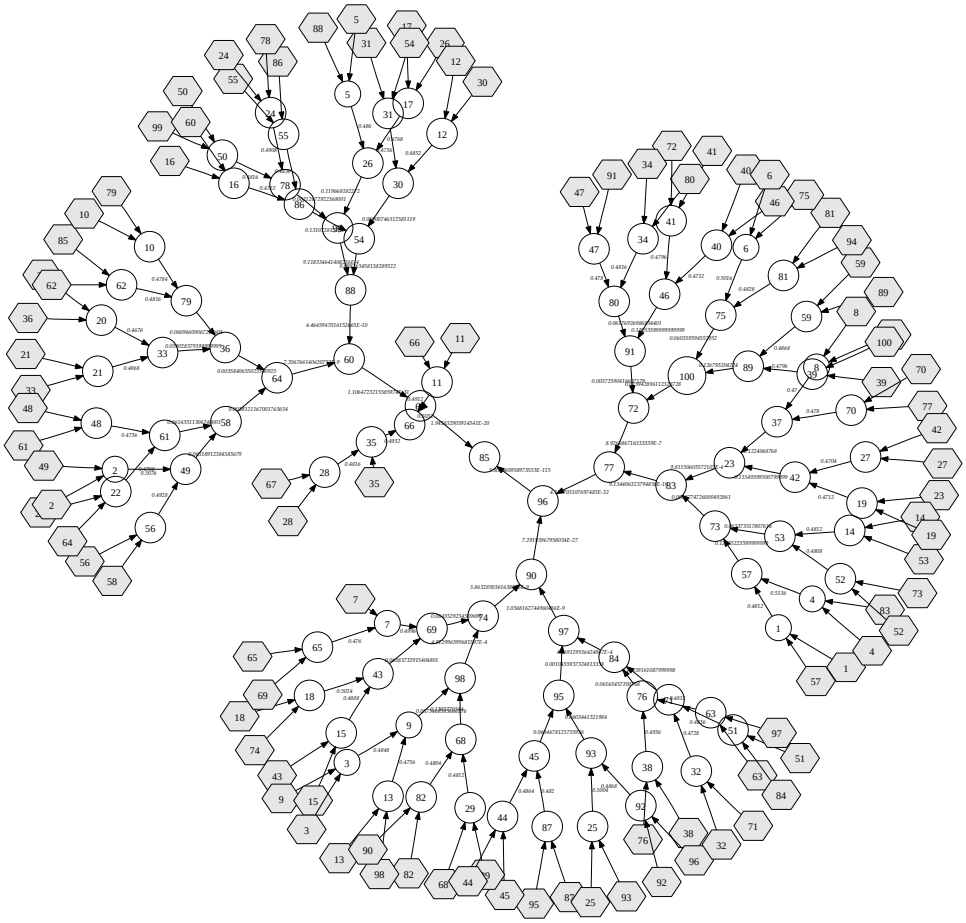


Figure 7.1: Partition of a randomly generated problem with 100 agents, with $p_1 = 0.4$ and $p_2 = 0.5$. Each virtual constraint weight between the sub-components of a representative agent is shown on its outgoing edge. Representative agent 85 is at the root of the partition hierarchy.

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.

Chapter 8

Conclusions

We presented two new distributed search algorithms for Distributed CSPs in this work. The algorithms are two-phase, where the first phase partitions the constraints network, and the second phase searches for solutions while exploiting the topological landscape created by the first phase.

The first part of both algorithms partitions the distributed constraints network into a binary tree of groups. The distributed partition algorithm uses a heuristic that selects to join neighbors that are strongly constrained, into groups. This is done concurrently at all levels of the hierarchy.

In DISHS, search is performed concurrently on disjoint parts of the global search space, by agents that constitute these parts. Agents form a hierarchy of groups and each group generates consistent partial solutions. Partial solutions are produced concurrently and are combined into consistent global solutions by agents that are higher in the hierarchy. The process stops when the top-level agent, the leader of all groups, reports either a solution or a failure.

In DESRS, concurrent independent backtracking search processes grow partial assignments along a hierarchy of agent groups, with each agent participating in multiple search processes. Stochastic choices for the order of assigning agents are taken after backtracking, securing the growing partial assignments from stalling in local minima.

The first algorithm, DISHS, did not provide satisfactory results. We speculated that this is an indication that some form of backtracking is necessary for good performance of DISCSP algorithms. We have also used DISHS to show that the group partitioning phase is vital, in the sense that reversing the neighbors preference ordering in the algorithm completely deteriorates the performance of the search phase.

To alleviate the shortcomings of the first algorithm, the second search algorithm, DESRS, was presented. In the new algorithm, 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.

In DESRS, the hierarchical structure of groups can be viewed 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.

We have also evaluated DESRS against the best performing concurrent search algorithm, CONCDB. 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 deadends discovered by other search processes. This option is in principle possible also for DESRS. This potential improvement of DESRS is left for a future study.

Finally, the group partitioning approach is not limited to distributed constraint satisfaction algorithms. We have generalized this approach, and touched upon other possible areas of its applications, such as social networks.

Appendix A

A Group Partitioning Messages Log

Messages received during group partitioning

```
Log of *received* messages.
  "Join<1> -> 3" means that agent 3 received Join<1> message
  (from agent 1).
  NoJoin and Search messages also have the source specified for clarity
  Components lists omit 0-level neighbors field, for brevity.
  Level of -1 is level "p".
------------------------------------------------------------------------

Leader<1, {(4,0.64) (9,0.66) (6,0.62) (3,0.62)}, {(1,N,0)}, true, 0> -> 1
    (level -1)
Leader<2, {(4,0.61) (7,0.6) (10,0.56)}, {(2,N,0)}, true, 0> -> 2
    (level -1)
Leader<3, {(9,0.6) (6,0.58) (1,0.62) (7,0.59) (5,0.63)}, {(3,N,0)}, true,
    0> -> 3 (level -1)
Join<1> -> 3 (level 0)
NoJoin<3> -> 1 (level 0)
1->3 JOIN ATTEMPT REPEATED ONCE
Leader<4, {(2,0.61) (9,0.62) (1,0.64) (7,0.6) (5,0.59)}, {(4,N,0)}, true,
    0> -> 4 (level -1)
Leader<5, {(4,0.59) (9,0.69) (3,0.63)}, {(5,N,0)}, true, 0> -> 5
   (level -1)
Join<5> -> 4 (level 0)
Join<4> -> 5 (level 0)
Components<4, {(4,N,0)}> -> 5 (level 0)
Components<5, {(5,N,0)}> -> 4 (level 0)
Leader<6, {(1,0.62) (3,0.58) (7,0.53) (10,0.6)}, {(6,N,0)}, true, 0> -> 6
    (level -1)
Join<3> -> 6 (level 0)
NoJoin<6> -> 3 (level 0)
Join<1> -> 6 (level 0)
NoJoin<6> -> 1 (level 0)
Join<1> -> 3 (level 0)
NoJoin<3> -> 1 (level 0)
1->3, 1->6, 3->6 JOIN ATTEMPTS REPEATED 126 TIMES
Leader<7, {(2,0.6) (4,0.6) (6,0.53) (3,0.59)}, {(7,N,0)}, true, 0> -> 7
    (level -1)
Join<6> -> 7 (level 0)
Leader<8, {}, {(8,N,0)}, true, 0> -> 8 (level -1)
Search<8> -> 8 (level -1)
Leader<9, {(4,0.62) (1,0.66) (3,0.6) (5,0.69)}, {(9,N,0)}, true, 0> -> 9
```

```
    (level -1)
Join<9> -> 3 (level 0)
NoJoin<3> -> 9 (level 0)
9->3 JOIN ATTEMPTS REPEATED 7 TIMES
Done<4, 4> -> 2 (level 0)
Done<4, 4> -> 4 (level 0)
Done<4, 4> -> 1 (level 0)
Done<4, 4> -> 5 (level 0)
Done<4, 4> -> 7 (level 0)
Done<5, 4> -> 4 (level 0)
Done<5, 4> -> 5 (level 0)
Done<4, 4> -> 9 (level 0)
Done<5, 4> -> 9 (level 0)
Done<5, 4> -> 3 (level 0)
Join<9> -> 3 (level 0)
NoJoin<3> -> 9 (level 0)
9->3 JOIN ATTEMPTS REPEATED 22 TIMES
Join<9> -> 3 (level 0)
Leader<10, {(2,0.56) (6,0.6)}, {(10,N,0)}, true, 0> -> 10 (level -1)
Join<3> -> 6 (level 0)
NoJoin<3> -> 9 (level 0)
Join<10> -> 2 (level 0)
Join<2> -> 10 (level 0)
NoJoin<6> -> 3 (level 0)
Join<1> -> 6 (level 0)
Join<9> -> 3 (level 0)
Components<10, {(10,N,0)}> -> 2 (level 0)
Components<2, {(2,N,0)}> -> 10 (level 0)
Done<2, 2> -> 2 (level 0)
Done<2, 2> -> 4 (level 0)
Done<2, 2> -> 7 (level 0)
NoJoin<3> -> 9 (level 0)
NoJoin<6> -> 1 (level 0)
Join<7> -> 6 (level 0)
Join<9> -> 3 (level 0)
Done<10, 2> -> 2 (level 0)
Done<2, 2> -> 10 (level 0)
NoJoin<3> -> 9 (level 0)
Components<6, {(6,N,0)}> -> 7 (level 0)
Components<7, {(7,N,0)}> -> 6 (level 0)
Done<10, 2> -> 10 (level 0)
Done<7, 6> -> 2 (level 0)
Done<7, 6> -> 4 (level 0)
Done<7, 6> -> 7 (level 0)
Done<7, 6> -> 3 (level 0)
Done<6, 6> -> 1 (level 0)
Join<3> -> 6 (level 0)
Leader<2, {(4,0.61) (6,0.6)}, {(2,N,1)}, false, 1> -> 2 (level -1)
Done<6, 6> -> 7 (level 0)
Join<9> -> 3 (level 0)
Done<6, 6> -> 10 (level 0)
Done<6, 6> -> 3 (level 0)
Join<1> -> 6 (level 0)
NoJoin<6> -> 3 (level 0)
NoJoin<3> -> 9 (level 0)
Leader<10, {(6,0.6)}, {(10,N,0)}, false, 1> -> 2 (level -1)
NoJoin<6> -> 1 (level 0)
Done<10, 2> -> 6 (level 0)
Join<1> -> 3 (level 0)
Join<3> -> 9 (level 0)
NoJoin<3> -> 1 (level 0)
Join<9> -> 3 (level 0)
```

```
Done<7, 6> -> 6 (level 0)
Join<1> -> 3 (level 0)
Components<3, {(3,N,0)}> -> 9 (level 0)
NoJoin<3> -> 1 (level 0)
Components<9, {(9,N,0)}> -> 3 (level 0)
Done<6, 6> -> 6 (level 0)
Done<3, 3> -> 1 (level 0)
Done<3, 3> -> 7 (level 0)
Done<3, 3> -> 5 (level 0)
Join<1> -> 3 (level 0)
Done<9, 3> -> 4 (level 0)
Done<3, 3> -> 9 (level 0)
Done<9, 3> -> 1 (level 0)
Done<9, 3> -> 5 (level 0)
Done<3, 3> -> 3 (level 0)
Done<3, 3> -> 6 (level 0)
NoJoin<3> -> 1 (level 0)
Done<9, 3> -> 3 (level 0)
Done<9, 3> -> 9 (level 0)
Join<1> -> 1 (level 0)
Components<1, {(1,N,0)}> -> 1 (level 0)
Done<1, 1> -> 4 (level 0)
Leader<5, {(3,0.4347)}, {(5,N,0)}, false, 1> -> 4 (level -1)
Leader<4, {(2,0.61) (6,0.6) (1,0.64) (3,0.62)}, {(4,N,1)}, false, 1> -> 4
    (level -1)
Done<1, 1> -> 1 (level 0)
Leader<1, {(4,0.64) (6,0.62) (3,0.4092)}, {(1,N,0)}, true, 1> -> 1
    (level -1)
Done<1, 1> -> 3 (level 0)
Leader<3, {(4,0.63) (6,0.34219999999999995) (1,0.62)}, {(3,N,1)}, false,
    1> -> 3 (level -1)
Done<1, 1> -> 9 (level 0)
Leader<9, {(4,0.42779999999999996) (1,0.66)}, {(9,N,0)}, false, 1> -> 3
    (level -1)
Join<3> -> 4 (level 1)
Join<4> -> 3 (level 1)
Components<3, {(9,N,0) (3,N,1)}> -> 4 (level 1)
Done<4, 5> -> 2 (level 1)
Done<4, 5> -> 4 (level 1)
Done<4, 5> -> 1 (level 1)
Join<1> -> 3 (level 1)
NoJoin<3> -> 1 (level 1)
Components<4, {(5,N,0) (4,N,1)}> -> 3 (level 1)
Done<3, 5> -> 4 (level 1)
Done<3, 5> -> 1 (level 1)
Done<4, 5> -> 3 (level 1)
Join<1> -> 3 (level 1)
NoJoin<3> -> 1 (level 1)
Done<3, 5> -> 3 (level 1)
Done<1, 1> -> 6 (level 0)
Leader<7, {(2,0.6) (4,0.6) (3,0.59)}, {(7,N,0)}, false, 1> -> 6
    (level -1)
Leader<6, {(2,0.6) (1,0.62) (3,0.58)}, {(6,N,1)}, false, 1> -> 6
    (level -1)
Join<6> -> 3 (level 1)
Join<2> -> 6 (level 1)
NoJoin<6> -> 2 (level 1)
Done<4, 5> -> 6 (level 1)
Done<3, 5> -> 6 (level 1)
Join<1> -> 6 (level 1)
NoJoin<6> -> 1 (level 1)
NoJoin<3> -> 6 (level 1)
```

```
Join<6> -> 2 (level 1)
Join<2> -> 6 (level 1)
Components<6, {(7,N,0) (6,N,1)}> -> 2 (level 1)
Done<2, 7> -> 2 (level 1)
Done<2, 7> -> 4 (level 1)
Join<1> -> 6 (level 1)
NoJoin<6> -> 1 (level 1)
Components<2, {(2,N,1) (10,N,0)}> -> 6 (level 1)
Done<6, 7> -> 2 (level 1)
Leader<2, {(5,0.61)}, {(2,N,1) (10,N,0)}, false, 2> -> 7 (level -1)
Done<6, 7> -> 4 (level 1)
Done<6, 7> -> 1 (level 1)
Done<6, 7> -> 3 (level 1)
Done<2, 7> -> 6 (level 1)
Join<1> -> 6 (level 1)
NoJoin<6> -> 1 (level 1)
Join<1> -> 1 (level 1)
Components<1, {(1,N,0)}> -> 1 (level 1)
Done<1, 1> -> 4 (level 1)
Done<1, 1> -> 1 (level 1)
Leader<1, {(7,0.62) (5,0.261888)}, {(1,N,0)}, true, 2> -> 1 (level -1)
Leader<4, {(1,0.64) (7,0.366)}, {(5,N,2) (4,N,1)}, false, 2> -> 5
    (level -1)
Done<1, 1> -> 3 (level 1)
Leader<3, {(1,0.4092) (7,0.34219999999999995)}, {(9,N,0) (3,N,1)}, false,
    2> -> 5 (level -1)
Join<1> -> 5 (level 2)
NoJoin<5> -> 1 (level 2)
1->5 JOIN ATTEMPTS REPEATED 59 TIMES
Done<6, 7> -> 6 (level 1)
1->5 JOIN ATTEMPTS REPEATED 82 TIMES
Done<1, 1> -> 6 (level 1)
1->5 JOIN ATTEMPTS REPEATED 41 TIME
Leader<6, {(1,0.62) (5,0.20531999999999997)}, {(7,N,2) (6,N,1)}, false,
    2> -> 7 (level -1)
Join<5> -> 7 (level 2)
Join<1> -> 5 (level 2)
NoJoin<5> -> 1 (level 2)
Join<7> -> 5 (level 2)
Components<5, {(9,N,0) (3,N,1) (5,N,2) (4,N,1)}> -> 7 (level 2)
Done<7, 9> -> 7 (level 2)
Done<7, 9> -> 1 (level 2)
Components<7, {(2,N,1) (7,N,2) (10,N,0) (6,N,1)}> -> 5 (level 2)
Done<5, 9> -> 1 (level 2)
Done<5, 9> -> 7 (level 2)
Join<1> -> 5 (level 2)
NoJoin<5> -> 1 (level 2)
Join<1> -> 1 (level 2)
Components<1, {(1,N,0)}> -> 1 (level 2)
Done<1, 1> -> 7 (level 2)
Leader<7, {(1,0.62)}, {(2,N,1) (7,N,2) (10,N,0) (6,N,1)}, false, 3> -> 9
    (level -1)
Done<1, 1> -> 1 (level 2)
Leader<1, {(9,0.16237056)}, {(1,N,0)}, true, 3> -> 1 (level -1)
Done<7, 9> -> 5 (level 2)
Done<5, 9> -> 5 (level 2)
Done<1, 1> -> 5 (level 2)
Leader<5, {(1,0.261888)}, {(3,N,1) (5,N,2) (9,N,3) (4,N,1)}, false, 3>
    -> 9 (level -1)
Join<1> -> 9 (level 3)
Join<9> -> 1 (level 3)
Components<1, {(1,N,0)}> -> 9 (level 3)
```

```
Done<9, 1> -> 9 (level 3)
Components<9, {(2,N,1) (3,N,1) (5,N,2) (9,N,3) (7,N,2) (6,N,1) (10,N,0)
    (4,N,1)}> -> 1 (level 3)
Done<1, 1> -> 9 (level 3)
Done<9, 1> -> 1 (level 3)
Done<1, 1> -> 1 (level 3)
Leader<9, {}, {(2,N,1) (3,N,1) (5,N,2) (9,N,3) (7,N,2) (6,N,1) (10,N,0)
    (4,N,1)}, false, 4> -> 1 (level -1)
Leader<1, {}, {(1,N,4)}, false, 4> -> 1 (level -1)
Search<1> -> 2 (level -1)
Search<1> -> 3 (level -1)
Search<1> -> 9 (level -1)
Search<1> -> 7 (level -1)
Search<1> -> 10 (level -1)
Search<1> -> 4 (level -1)
Search<1> -> 1 (level -1)
Search<1> -> 5 (level -1)
Search<1> -> 6 (level -1)
```

Appendix B

Operations on Complete Binary Tree

Algorithm B.1: CT-ACCESS(T, cap, r, i): Access the i^{th} element of a complete binary tree.

Input : tree T, last row capacity cap, last row count r, node pre-order index i
Output: requested node is returned
Locals : P $\leftarrow$ NIL

```
1  while i ≠ 0 do
2      P ← T
3      i ← i − 1
4      cap ← cap/2
5      r_left ← min{r, cap}
6      left ← cap − 1 + r_left
7      if i < left then
8          T ← T_left
9      else
10         T ← T_right
11         i ← i − left
12         r ← r − r_left
13 return ⟨T, P⟩
```

Algorithm B.2: CT-REMOVE(T, cap, r, i): Remove the i^{th} element from a complete binary tree.

Input : tree T, last row capacity cap, last row count r, node pre-order index i
Output: requested node is removed from the tree, and its value is returned

```
1  ⟨P_S, S⟩ ← CT-ACCESS(T, cap, r, i )
2  ⟨P_Q, Q⟩ ← CT-ACCESS(T, cap, r, cap +r-2)
3  Q_children ← S_children
4  P_S children[S] ← Q
5  P_Q children[Q] ← NIL
6  r ← r − 1
7  if r = 0 then
8      r ← cap ← ⌊cap/2⌋
9  return ⟨S_info, cap, r⟩
```

Algorithm B.3: CT-ADD(T, cap, r, info): Add an element to a complete binary tree.

Input : tree T, last row capacity cap, last row count r, node value info
Output: requested node is returned
Locals : P ← NIL, S ← NEW-NODE(info)

```
1  r ← (r mod cap) + 1
2  if r = 1 then
3      cap ← 2 · cap
4  T' ← T,   r' ← r,   cap' ← cap
5  if T = NIL then
6      T' ← S
7      cap ← 1
8  else
9      while T ≠ NIL do
10         P ← T
11         cap ← cap/2
12         if r ≤ cap then
13             T ← T_left
14         else
15             T ← T_right
16             r ← r − cap
17     P_children ← P_children ∪ {S}
18 return ⟨T', cap', r'⟩
```

Bibliography

[1] Slim Abdennadher and Hans Schlenker. Nurse scheduling using constraint logic programming. In *Proceedings of the Eleventh Annual Conference on Innovative Applications of Artificial Intelligence*, pages 838–843, Orlando, Florida, USA, July 1999.

[2] Andrew Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.

[3] John Barnes. Class and committees in a Norwegian island parish. *Human Relations*, 7:39–58, February 1954.

[4] Christian Bessière, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In *Notes of the IJCAI'01 Workshop on Distributed Constraint Reasoning*, pages 9–16, Seattle, Washington, USA, 2001.

[5] 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.

[6] Ismel Brito and Pedro Meseguer. Distributed forward checking. In *Principles and Practice of Constraint Programming — CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 801–806, Kinsale, Ireland, September 2003.

[7] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, May 2003. ISBN 1-558-60890-7.

[8] Rina Dechter and Judea Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, December 1987.

[9] Matthew Ginsberg. Dynamic backtracking. *Artificial Intelligence Research*, 1:25–46, August 1993.

[10] Youssef Hamadi, Christian Bessière, and Joël Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 219–223, Brighton, UK, August 1998.

[11] Kazuyoshi Honda and Fumio Mizoguchi. Constraint-based approach for automatic spatial layout planning. In *Proceedings of the Eleventh Conference on Artificial Intelligence for Applications*, pages 38–45, Los Angeles, California, USA, February 1995.

[12] Eliezer Kaplansky and Amnon Meisels. Distributed personnel scheduling — negotiation among scheduling agents. *Annals of Operations Research*, 2006. To appear, `http://www.cs.bgu.ac.il/~am/My_Papers/DisETP_Annals_OR.pdf`.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

[14] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, April 1997. ISBN 1-558-60348-4.

[15] Amnon Meisels. Distributed constraints: Algorithms, performance, communication. In *CP-2004: Tutorials*, Toronto, Canada, September 2004.

[16] 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*, pages 86–93, Bologna, Italy, July 2002.

[17] Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1–2):81–109, March 1996.

[18] Helmut Simonis. Sudoku as a constraint problem. In *Proceedings of the Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, Barcelona, Spain, October 2005.

[19] Barbara Smith and Martin Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1–2):155–181, March 1996.

[20] Gadi Solotorevsky, Ehud Gudes, and Amnon Meisels. Modeling and solving distributed constraint satisfaction problems (DCSPs). In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 561–562, Cambridge, Massachusetts, October 1996.

[21] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, August 1993. ISBN 0-127-01610-4.

[22] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1–2):139–168, September 1996.

[23] Makoto Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer Series on Agent Technology. Springer-Verlag, Berlin, 2001.

[24] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes In Computer Science*, pages 88–102, Cassis, France, 1995.

[25] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, June 2000.

[26] Makoto Yokoo and Katsutoshi Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of the Third International Conference on Multi Agent Systems*, pages 372–379, La Villette, Paris, France, July 1998.

[27] Makoto Yokoo, Toru Ishida, Edmund Durfee, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pages 614–621, Yokohama, Japan, June 1992.

[28] Makoto Yokoo, Edmund Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Data and Knowledge Engineering*, 10(5):673–685, September 1998.

[29] Weixiong Zhang and Lars Wittenburg. Distributed breakout revisited. In *Eighteenth national conference on Artificial intelligence*, pages 352–357, Edmonton, Alberta, Canada, 2002.

[30] Weixiong Zhang, Guandong Wang, and Lars Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Workshop on Probabilistic Approaches in Search, AAAI-2002*, pages 53–59, Edmonton, Alberta, Canada, July 2002.

[31] Roie Zivan and Amnon Meisels. Concurrent backtrack search on DisCSPs. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Symposium Conference*, pages 776–781, Miami Beach, Florida, USA, May 2004.

[32] 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*, pages 782–787, Toronto, Canada, January 2004.

[33] Roie Zivan and Amnon Meisels. Concurrent search for distributed CSPs. *Artificial Intelligence*, 170(4–5):440–461, April 2006.