A Decentralized Variable Ordering Method for Distributed Constraint Optimization

Anton Chechetka Katia Sycara

CMU-RI-TR-05-18

May 2005

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Many different multi-agent problems, such as distributed scheduling can be formalized as distributed constraint optimization. Ordering the constraint variables is an important preprocessing step of the ADOPT algorithm [1], the state of the art method of solving distributed constraint optimization problems (DCOP). Currently ADOPT uses depth-first search (DFS) trees for that purpose. For certain classes of tasks DFS ordering does not exploit the problem structure as compared to pseudo-tree ordering [2]. Also the variables are currently ordered in a centralized manner, which requires global information about the problem structure. We present a variable ordering algorithm, which is both decentralized and makes use of pseudo-trees, thus exploiting the problem structure when possible. This allows to apply ADOPT to domains, where global information is unavailable, and find solutions more efficiently. The worst-case pseudo-tree depth resulting from our algorithm is $\sqrt{2kn}$, where n is the number of variables, and k is maximum block size in constraint graph. The algorithm has space complexity of $O(kn)$ and time complexity of $O(n+|E|+k^{\frac{3}{2}}n^{\frac{1}{2}})$, where E is the set of edges in a constraint graph.

Contents

1 Introduction **1**

2 Acyclic Case **3**
2.1 The Algorithm 3
2.2 Algorithm Properties 5
2.2.1 Correctness. 5
2.2.2 Complexity 8

3 Limited-block Case **8**
3.1 Algorithm Overview 9
3.2 Block Discovery 10
3.3 Tree Root Selection 11
3.4 Algorithm Properties 14
3.5 Complexity 16

4 Evaluation **17**
4.1 Ordering algorithm cost 17
4.2 ADOPT performance 18

5 Conclusion **18**

6 Acknowledgements **19**

1 Introduction

The distributed constraint optimization problem (DCOP) [3] is increasingly used as an underlying framework for modelling agents coordination problems. Sensor networks [4], distributed scheduling [5], military unmanned aerial vehicles teams [6] all are applications, where DCOP is or can be used to coordinate the decisions of autonomous agents.

DCOP consists of variables, each having its own discrete finite domain. Every variable's value is controlled by exactly one agent. Like the majority of the work in the field, we assume that an agent controls only one variable, although the situation of one agent controlling several variables is also present in the literature [7]. The agents must coordinate in order to minimize the global cost function, which depends on the variables. The cost function is modeled as a set of valued constraints, and every agent has knowledge only about the constraints depending on its variable. Therefore, DCOP is a generalization of distributed constraint satisfaction problem (DisCSP) [8].

Both DCOP and DisCSP stem from the extensively studied constraint satisfaction problem (CSP) [9], and the algorithms to solve the distributed problems are often obtained by adapting the existing CSP algorithms to work in a decentralized manner. Several additional factors should be taken into account when performing such adaptation. First, instead of a single processing unit, in DCOP there are multiple processing units, thus it is desirable to take advantage of this increased computing power. Second, the usual assumption is that the communication between agents can be performed only locally, i.e. a message can be passed only between agents that share a constraint. This imposes a requirement of minimizing the communication cost of the algorithm.

The connectivity between the agents can be represented in form of a constraint graph. The vertices of the graph are agents, and two agents have an edge between them, iff they share a constraint.

Recently an algorithm for solving DCOP, ADOPT [1] was proposed. It performs a systematic backtracking with agents acting asynchronously. To achieve that, the agents are ordered a priori in a tree such that the agents in different branches of the tree do not share any constraints. ADOPT itself does not address the process of ordering and regards it as a preprocessing step. Currently a centralized algorithm that performs depth-first traversal of the constraint graph, is used for that purpose.

The higher the agent is in the tree, the higher priority it has for choosing its variable value. The agents in different branches can act independently of each other. It is known [1] that the performance of the algorithm crucially depends on variable ordering, thus making important the task of constructing the tree.

It is hard to find an optimal tree for ADOPT without actually solving the DCOP, given that the variables may have domains of different size, different constraints may be more or less restrictive, etc. However, tree depth is a good approximation of the optimality criterion, because it captures both the issue of parallelism - generally, the smaller the depth the more branches the tree has; and information travel time - the smaller the depth the shorter the path of the

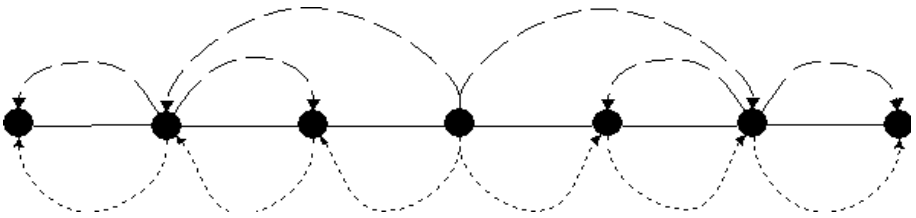

Figure 1: 7-nodes chain constraint graph. Dotted arrows mark DFS tree ordering. Dashed arrows - pseudo-tree ordering.

information about variable assignments travelling from root to leaves.

Finding the minimum depth DFS tree of the graph is NP-complete problem, so different heuristics, such as max-degree [10], are usually employed to obtain an approximate solution. Distributed algorithm for constructing DFS trees can be found in [11].

For certain classes of tasks, however, restricting the possible orderings to DFS trees can have strong negative impact on ADOPT performance as compared to pseudo-trees [2]. Consider for example the collective choice of firing positions by unmanned helicopters, presented in [6]. The constraint graph for this task has a chain structure, with every agent sharing constraints with its left and right neighbors (Fig. 1). The DFS tree for this graph has depth linear in number of variables. Optimal pseudo-tree for the same graph has logarithmic depth. Therefore, the former case results in longer time to find the DCOP solution than the latter.

The centralized iterative algorithm for constructing pseudo-trees was presented in [2], and further developed in [12] to strengthen tree depth guarantees. It was shown for certain classes of CSP to yield better performance of the backtracking algorithm than the DFS orderings. Because of the similarity of the CSP and DCOP problems and algorithms, one can expect similar results for DCOP. The drawback of these methods is that they need complete constraint graph connectivity information. This requirement does not allow to apply ADOPT to domains in which information about the constraint graph cannot be gathered together and processed by a single unit. Sensor networks [6] is an example of such domain.

In this paper we present an algorithm which is both decentralized and makes use of pseudo-trees. The former property enables one to solve DCOP problems without having to process any global information, and the latter increases the efficiency of the solution search if the problem structure allows it. Our algorithm is a decentralized modification of the general iterative algorithm from [2]. Unlike [12], it takes into account not only the depth of the resulting pseudo-tree, but also the maximum message travel time from root to leaves given the local communication assumptions. It yields an optimal pseudo-tree in terms of message travel time for acyclic graphs and results in tree depth no greater than DFS traversal for general graphs.

2 Acyclic Case

2.1 The Algorithm

Suppose that the constraint graph $X =< V, E >$ is acyclic. Let all the edges have unit length. Denote N the set of agent x's neighbors. We will use the terms *agent*, *node*, and *vertice* interchangeably. We will also substitute pseudo-tree for tree, where it does not cause confusion with DFS trees. When speaking of subgraphs $X(V_i \subseteq V) \equiv X_i \subseteq X$, we will assume that E_i contains all the edges from E, which pass between vertices from V_i and does not contain any other edges.

Definition 1 *For agent x and $X_0 \subseteq X$*

$$AgentDepth(x, X_0) \equiv \max\{(AgentDepth(x_i, X(V_0 \setminus x)) + 1, \forall x_i \in V_0 \cap N), 0\} \quad (1)$$

Definition 2 *For constraint graph X*

$$GraphDepth(X) \equiv \min_{x \in X} AgentDepth(x, X) \quad (2)$$

Observe that $AgentDepth(x, X)$ is the maximum length of a simple path in X beginning in x.

The tree building algorithm is as follows. First, select a node x_1 such that $AgentDepth(x_1) = GraphDepth(X)$. Assign this node to be a root of the tree. Second, remove x_1 from the graph (this does not mean re-formulating the optimization problem - the node and corresponding edges are only removed from consideration by the tree-building algorithm). For every connected subgraph formed after the removal repeat the algorithm recursively, and set the next-level roots to be children of x_1 in the tree ordering. This algorithm can be executed by running the locally communicating asynchronous agents, so that every agent controls one node (Fig. 2).

The result of the algorithm is that every agent knows the local tree information.

Definition 3 *Local tree information for agent x is*

- *ancestors $\subset$ V, an ordered set such that ancestors[1] is the root of the tree, ancestors[k] is the parent of ancestors[k+1], and ancestors[last] is the parent of x.*
- *children $\subset$ V, an ordered set containing all the children of x.*
- *ancestors_routes $\subset$ N, an ordered set such that ancestors_routes[k] is the first node (not counting x itself) of the simple path from x to ancestors[k].*
- *ancestors_distances, an array of positive integers such that ancestors_distances[k] is the length of the simple path from x to ancestors[k].*

```
1:  receive AgentDepth(x_i, X \ x) from all x_i ∈ N except one x_k;
2:  send AgentDepth(x, X \ x_k) to x_k;
3:  receive AgentDepth(x_k, X \ x) from x_k;
4:  send AgentDepth(x, X \ x_i) to all x_i ∈ N \ x_k;
5:  send AgentDepth(x, X) to all x_i ∈ N;
6:  receive AgentDepth(x_i, X) from all x_i ∈ N;
7:  L = 0;
8:  while |N| > 0
9:    if ∀x_i ∈ N AgentDepth(x, X) < AgentDepth(x_i, X)
10:     send "child x" to ancestors[last] via ancestors_routes[last];
11:     send "ancestor x, distance = 1" to all x_i ∈ N;
12:     for ∀x_i ∈ N
13:       receive "child y_i";
14:       append y_i to children;
15:       append x_i to children_routes;
16:     end for;
17:     exit;
18:   else
19:     L = L + 1;
20:     receive "ancestor a, distance = dist" from x_a ∈ N;
21:     append a to ancestors;
22:     append x_a to ancestors_routes;
23:     append dist to ancestors_distances;
24:     send "ancestor a, distance = dist+1" to all x_i ∈ N \ x_a;
25:     if a ≡ x_a
26:       remove a from N;
27:     else
28:       receive AgentDepth(x_a, X_L \ x) from x_a;
29:       receive AgentDepth(x_k, X_L) from x_k;
30:       send AgentDepth(x, X_L) to x_k;
31:     end if;
32:     send AgentDepth(x, X_L \ x_i) to all x_i ∈ N \ x_k;
33:     send AgentDepth(x, X_L) to all x_i ∈ N \ x_k;
34:     receive AgentDepth(x, X_L) from all x_i ∈ N \ x_k;
35:   end if;
36: end while;
```

Figure 2: Acyclic graph algorithm for agent x

- *children_routes* $\subset N$, *an ordered set such that children_routes[k] is the first node (not counting x itself) of the simple path from x to children[k].*

On line (1:) "all $x_i \in N$ except one x_k" means that an agent receives $|N|-1$ messages from its neighbors, and the remaining agent whose message was not received among these $|N|-1$ messages is denoted x_k. The receiver does not select x_k beforehand. $|N|-1$ values of neighbors' *AgentDepth* are enough to compute $AgentDepth(x, X \setminus x_k)$, and x does that without waiting for the $|N|$'th value. If the agents were to wait for messages from all their neighbors, communication would never be started and they all would be locked at line (1:).

Every cycle iteration agent x calculates its metric *AgentDepth* for the remaining subgraph and checks if it is the minimal value within that subgraph (line 9:). If it is, and the tie (if there is one) is broken in this agent's favor, then it becomes a node of depth L in the tree, reports this fact to its parent (10:), notifies the subgraphs (11:), waits for the information about its children (12:-16:) and exits the cycle. Otherwise it increases its L value (19:), receives and records the information about the next ancestor (20:-23:), updates the *AgentDepth* values that might change (25:-34:), and repeats the process for a smaller subgraph.

If two agents have the same *AgentDepth* equal to a minimum value, the tie is broken in favor of the agent which is closer to the previous level ancestor. It will be shown further, that a tie between more than two agents is impossible, thus the tiebreaking rule can always be employed, except for the first iteration of the algorithm, when there is no previous ancestor. In the latter case the tie is broken using agent names.

To reduce the communication load, the information about the *AgentDepth*'s, which cannot change with the removal of a given node, is not communicated. This means that instead of repeating (1:-6:) for each iteration, agents use (28:-34:) instead.

2.2 Algorithm Properties

2.2.1 Correctness.

The following two theorems show that for a given agent x it is sufficient to compare its *AgentDepth* value only with the *AgentDepth* values of x's neighbors to find out whether $AgentDepth(x, X) = GraphDepth(X)$.

Theorem 1 *For any acyclic graph X, if there are 2 nodes* x_1 *and* x_2 *such that* $AgentDepth(x_1, X) = AgentDepth(x_2, X) = GraphDepth(X)$*, then* x_1 *and* x_2 *are neighbors. No more than 2 nodes satisfy the requirement* $AgentDepth(x, X) = GraphDepth(X)$.

Proof: First observe that the second part of the statement follows directly from the first part and the graph being ayclic. Indeed, if there are 3 or more nodes in a tie, they must all be neighbors with each other, thus forming a cycle.

The proof of the first part is as follows: suppose $\exists x_1, x_2 : AgentDepth(x_1, X) = AgentDepth(x_2, X) = GraphDepth(X)$ and $x_1, x_{i_1}, \ldots, x_{i_k}, x_2$ is a simple path

from x_1 to x_2. By definition of *AgentDepth*,

$$\begin{array}{l} AgentDepth(x_{i_1}, X(V \setminus x_1)) \leq AgentDepth(x_1, X) - 1 \\ AgentDepth(x_{i_1}, X(V \setminus x_{i_2})) \leq AgentDepth(x_1, X) - dist(x_2, x_{i_1}) \end{array} \tag{3}$$

(note that $x_{i_2} = x_2$ is possible), and because $(V \setminus x_1) \cup (V \setminus x_{i_2}) \equiv V$,

$$\begin{array}{c} AgentDepth(x_{i_1}, X) \leq \\ \max\{AgentDepth(x_1, X) - 1, AgentDepth(x_1, X) - dist(x_2, x_{i_1})\} \\ < GraphDepth(X), \end{array} \tag{4}$$

a contradiction. Therefore, the assumption is incorrect. □

Theorem 2 *For any acyclic graph X, if for all neighbors x_i of a given node x holds $AgentDepth(x, X) \leq AgentDepth(x_i, X)$, then $AgentDepth(x, X) = GraphDepth(X)$.*

Proof: Suppose $AgentDepth(x, X) > GraphDepth(X)$. Let x_{min} be the node such that $AgentDepth(x_{min}, X) = GraphDepth(X)$, and $x, x_1, \ldots, x_k, x_{min}$ is a simple path from x to x_{min}. By definition of *AgentDepth*,

$$\begin{array}{l} AgentDepth(x_1, X(V \setminus x)) \leq AgentDepth(x, X) - 1 \\ AgentDepth(x_1, X(V \setminus x_2)) \leq AgentDepth(x_{min}, X) - dist(x_{min}, x_1) \leq \\ \leq GraphDepth(X) - 1 < AgentDepth(x, X) - 1 \end{array} \tag{5}$$

therefore,

$$\begin{array}{l} AgentDepth(x_1, X) \leq AgentDepth(x, X) - 1 \Rightarrow \\ \Rightarrow AgentDepth(x_1, X) < AgentDepth(x, X) \end{array} \tag{6}$$

a contradiction. □

The following result enables one to think of the graph metric $GraphDepth(X)$ as an upper bound of the resulting tree depth.

Theorem 3 *For any acyclic graph X, the result of the algorithm execution is a tree of depth no greater than $GraphDepth(X)$*

Proof: It is sufficient to prove that after removal of node x such that

$$AgentDepth(x, X) = GraphDepth(X)$$

any of the resulting connected subgraphs X_i has $GraphDepth(X_i) \leq GraphDepth(X) - 1$. The number of subgraphs is equal to the number of x's neighbors. For each neighbor x_i of x (N_i is the set of x_i's neighbors in X)

$$\begin{array}{l} AgentDepth(x_i, X_i) = AgentDepth(x_i, X(V \setminus x)) \leq \\ \leq AgentDepth(x, X) - 1 = GraphDepth(X) - 1, \end{array} \tag{7}$$

therefore $GraphDepth(X_i) \leq GraphDepth(X) - 1$. □

The next result shows that giving up the requirement of parent and child being neighbors in the constraint graph does not lead to a worse solution in terms of maximum message travel time from root to leaves.

Theorem 4 *For any acyclic graph X, if in the resulting tree x_1 is a root, x_i is a parent of x_{i+1}, $i = \overline{1, k-1}$, x_k is a leaf, then*

$$\sum_{i=1}^{k-1} dist(x_i, x_{i+1}) \leq GraphDepth(X)$$

Note that the sum under consideration is the number of "hops" the message sent from x_1 to x_k encounters, if it is routed through the consecutive descendants of the resulting tree.

Proof: It is sufficient to prove that if x is a root of the tree and x_1 is its child, then $AgentDepth(x, X) \geq dist(x, x_1) + AgentDepth(x_1, X(V \setminus x))$. Suppose $dist(x, x_1) + AgentDepth(x_1, X(V \setminus x)) > AgentDepth(x, X)$. There exists a simple path from x to x_1, denote it $x, x_{i_1}, \ldots, x_{i_k}, x_1$. Because

$$AgentDepth(x_1, X(V \setminus x_{i_k})) \leq AgentDepth(x, X) - dist(x, x_1), \tag{8}$$

and

$$AgentDepth(x_{i_k}, X(V \setminus \{x, x_1\})) \leq AgentDepth(x_1, X(V \setminus x)) - 1, \tag{9}$$

we get

$$\begin{gathered} AgentDepth(x_{i_k}, X(V \setminus x)) = \\ = \max\{(AgentDepth(x_1, X(V \setminus x)) - 1), AgentDepth(x, X) - dist(x, x_1) + 1\} \leq \\ \leq AgentDepth(x_1, X(V \setminus x)) \end{gathered} \tag{10}$$

and because $dist(x, x_{i_k}) < dist(x, x_1)$, even in case $AgentDepth(x_{i_k}, X(V \setminus x)) = AgentDepth(x_1, X(V \setminus x))$, the tiebreaking rule would choose x_{i_k} over x_1 as a root for the subgraph, therefore the assumption is impossible.□

Theorem 5 *For any acyclic graph X the depth of the resulting tree is less than $\sqrt{2|V|}$*

Proof: Consider a resulting tree ordering with depth h. There are agents $x_0, \ldots, x_h$ such that x_k is a parent of x_{k+1}. When the root of the tree was selected, the constraint graph was divided into the subgraph $X_1 \supset \{x_1, \ldots, x_h\}$ and at least h agents not contained by X_1. The proof of this fact is as follows. By Theorem 3,

$$AgentDepth(x_0, X) \geq h \tag{11}$$

Let $x_0, x_{i_1}, \ldots, x_{i_k}, x_1$ be the simple path from x_0 to x_1. Then

$$AgentDepth(x_{i_1}, X(V \setminus x_0)) \leq AgentDepth(x_0, X) - 1 \tag{12}$$

If the assumption $|V(X_1)| > |V(X)| - h$ holds, then

$$AgentDepth(x_{i_1}, X(V \setminus (N_{i_1} \setminus x_0))) < h \tag{13}$$

Therefore, from (12) and (13)

$$AgentDepth(x_{i_1}, X) \leq h - 1 < AgentDepth(x_0, X) \tag{14}$$

a contradiction with $AgentDepth(x_0, X) = GraphDepth(X)$, and the assumption $|V(X_1)| > |V(X)| - h$ is wrong.

Applying this reasoning recursively, conclude that the total number of nodes in the graph

$$|V| \geq 1 + \sum_{0}^{h} k = 1 + \frac{h(h+1)}{2} \tag{15}$$

therefore

$$h < \sqrt{2|V|} \tag{16}$$

□

2.2.2 Complexity

The amount of memory required for a given agent x is $O(|N|+h(X)) = O(|N|+\sqrt{|V|})$, where h is the resulting pseudo-tree ordering depth. It is clear from the fact that the agent should remember the information about its ancestors (there can be at most $h(X) - 1$ of them) and children (at most $|N|$). Compared to a DFS tree, where local memory cost does not depend on the total number of nodes in the graph, but only on the number of neighbors, one can conclude that in our algorithm routing memory is traded off for the tree depth. In terms of the asymptotical memory requirement, in general $O(|N| + \sqrt{|V|})$ is equivalent to $O(|V|)$.

We estimate the time required to run the algorithm using the popular metric of synchronous *cycles* [13]. A cycle is defined as an event of all the agents sending all the messages that are ready and receiving all the messages that were sent during the previous cycle. It takes $O(GraphDepth(X)) = O(|V|)$ cycles to propagate the nodes metrics information along the graph. Because the information about paths depths is reused, after selection and removal of the ancestor node, to select the next level ancestor one needs to propagate the information about the ancestor only one step further than the node which would be the next level ancestor, which requires $dist(ancestor_i, ancestor_{i+1})+1$ cycles and the same number of cycles to propagate the child information back to the $ancestor_i$ from $ancestor_{i+1}$. Because

$$\sum dist(ancestor_i, ancestor_{i+1}) \leq GraphDepth(X), \tag{17}$$

total number of cycles needed is $O(GraphDepth(X)) = O(|V|)$.

3 Limited-block Case

This is a generalization of the acyclic case, allowing cycles of limited size to exist in the constraint graph.

```
37: m = receiveMessage(); prev = m.from; append x to m.stack;
38: for i = 1:|m.blocks|
39: if ((m.blocks[i] \ m.from) ∩ neighbors ≠ ∅)
40:     for k=m.blocks[i].top:|m.stack| add m.stack[k] to m.blocks[i];
41: end for; end if; end for;
42: for i = 1:(|m.stack|-2) if (m.stack[i] ∈ neighbors)
43:     for k=i:|m.stack| add m.stack[k] to new_block; end for;
44:     add new_block to m.blocks; break;
45: end if; end for;
46: ∀ i≠j: (m.blocks[i] ∩ m.blocks[j] > 1) merge(m.blocks[i], m.blocks[j]);

47: candidates = neighbors \ ((∪ m.blocks) ∪ m.stack);
48: while candidates ≠ ∅
49:     if (|m.stack| + ∑_i |m.blocks[i]| > MAX_BLOCK) break; end if;
50:     pick next ∈ candidates; candidates = candidates \ next;
51:     send m to next;                  //go one step deeper
52:     mNew = receiveMessage();    //receive control back
53:     (∀ i: mNew.blocks[i].top = |mNew.stack|) remove mNew.blocks[i];
54:     if mNew.blocks ≠ m.blocks        //blocks have been updated
55:     add next to BlockChildren; parent = prev; end if;
56:     m = mNew; candidates = candidates \ ((∪ m.blocks) ∪ m.stack);
57: end while;

58: remove m.stack[last]; //un-append this agent
59: send m to prev;       //return control up one level in DFS tree
```

Figure 3: Block discovery algorithm for agent x. *block.top* denotes the position of the highest block's element in the stack

Definition 4 *In graph theory a block is a maximal connected subgraph without a cutvertex*[1] *[14]*

Each block is either an isolated vertex, a bridge (two vertices and an edge between them), or a maximal 2-connected[2] subgraph. We will refer to only 2-connected subgraphs as blocks, disregarding the first two degenerate cases. It is known [14] that two blocks intersect in at most one vertex, that is a cutvertex of the graph.

From now on assume that the maximum block size in the constraint graph is k. This also limits the maximum simple cycle length to be no more than k.

3.1 Algorithm Overview

The algorithm proceeds in two major stages. During preliminary stage each agent discovers its block configuration. The main stage is the iterative process

[1]Cutvertex is a vertex that separates two other vertices in a connected subgraph

[2]Graph G is k-connected iff $\forall V_1 \subset V(G), |V_1| < k \;\; G - V_1$ is connected [14]

```
function blocksOrder()
60: initAgentDepths(NC, C);
61: while (true)
62:   if (AgentDepth(x, X) < AgentDepth(x_i, X) ∀x_i ∈ NC ∪ (∪_j C_j))
63:     MakeMyselfRoot(); exit;
64:   else //I am not a new root, propagate the new root information
65:     if received "child y", forward it via ancestors_routes[last]; end if;
66:     receive list from y ∈ N ; //list of newly-ordered ancestors
67:     append list to ancestors; set ancestor_routes[a] = y ∀ a ∈ list;
68:     if (∃C_i ∈ C : list[1] ∈ C_i) //I'm in the same block as root
69:       orderUsingDFS(C_i, list); exit;
70:     else if (∃C_i ∈ C : ancestors_list[last] ∈ C_i); //not in the same block as root,
71:       updateDepthsAndStructure(C_i, list); //but block configuration has changed
72:     else //not a root and block configuration hasn't changed
73:       updateDepths(list);
74: end if; end if; end while;
end blocksOrder;

function MakeMyselfRoot()
80:   send "child x" to ancestors[last] via ancestor_routes[last];
81:   send list=[x] to NC ∪ (∪_{∀C_j ∈ C} child(C_j));
82:   (for ∀C_j ∈ C) append child(C_j) to children and children_routes; end for;
83:   for ∀x_i ∈ NC
84:     receive "child y_i" from x_i;
85:     append y_i to children and x_i to children_routes;
86:   end for;
end MakeMyselfRoot;
```

Figure 4: Limited-block algorithm for agent x

of selecting nodes for several top levels of the hierarchy, removing them from consideration and considering resulting subgraphs, much like the acyclic case algorithm.

3.2 Block Discovery

The first stage of the algorithm is to discover the block configuration of every agent.

Definition 5 *Block configuration of the agent x is the following data:*

- *A set $NC \in N$ of neighbors, which do not belong to the same block as x.*
- *A set $C = \{C_1, \ldots, C_k\}$ of blocks, to which x belongs. $C_i = \{x_{i_1}, \ldots, x_{i_{|C_i|}}\}$ (all the block members are included, not only the neighbors of x)*
- *$\forall x_j \in C_i \setminus x$ $parent(x_j) \in N$, $BlockChildren(x_j) \subset N$ - the parent and children of x in the DFS spanning tree for C_i rooted at x_j.*

- $\forall C_i$ *child*$(C_i) \in N$ *the child of* x *in the DFS spanning tree for* C_i *rooted at* x*. (*x *cannot have more than one child in this tree).*

This information can be obtained using the multiple distributed depth-limited (no more than k+1) DFS traversals of the constraint graph, one traversal per agent. The traversals can be executed in parallel. Fig. 3 contains the algorithm pseudocode (for simplicity the algorithm shown is for only one traversal, started in a pre-determined agent). It performs limited-depth DFS traversal with currently known blocks structure (m.blocks) included in every message. Although it is not shown in the code, every node that is recorded in the message is accompanied by its depth in the traversal stack. The depth in the stack is equivalent to the distance of the given node from root within a block. Every message also includes current stack (m.stack) of the traversal. The blocks are merged (46:) whenever it is possible to determine that two blocks in a message are different subsets of the same block. If a block is found not to contain the root of the traversal (53:), it is removed from the message.

Because the maximum size of a message is $O(k)$ (49:), simultaneous traversals require no more than $O(nk)$ memory on each agent.

Definition 6 *The distance between* $x_1, x_2 \in X$*, denoted* $dist(x_1, x_2, X)$ *is the length of the simple path such that the subpaths between neighboring cutvertices* $x_{i_k}, x_{i_{k+1}}$ *are selected according to the DFS ordering of the corresponding block rooted at* x_{i_k}

Note that in general $dist(x_i, x_j, X) \neq dist(x_j, x_i, X)$.

3.3 Tree Root Selection

Once the block configuration has been determined, one can use essentially the same iterative algorithm as in acyclic case to construct a tree. (see Fig. 4, 5, 6 and the following definitions).

We assume that each agent has a prioritization on its neighbors, and selects the next node to visit in a DFS traversal (50:) so that it has the highest priority among all the avaliable neighbors. Further assume that this prioritization does not change over time. Position of a node in a list of neighbors is an example of such prioritization.

Definition 7 *For agent* x*, graph* X *and set* $NCS \subseteq NC(x, X), CS \subseteq C(x, X))$

$$\begin{array}{c} AgentDepth(x, NCS, CS, X) \equiv \\ \max\{0, (AgentDepth(x_i, NC_i \setminus x, C_i, X) + 1, \forall x_i \in NCS), \\ (AgentDepth(x_{i_j}, NC_{i_j}, C_{i_j} \setminus C_i, X) + dist(x, x_{i_j}, X), \\ \forall x_{i_j} \in C_i \setminus x, \forall C_i \in CS)\} \end{array} \tag{18}$$

Definition 8 *For any agent* $x \in X$*, where* X *is a connected graph*

$$AgentDepth(x, X) \equiv AgentDepth(x, NC(x, X), C(x, X), X) \tag{19}$$

function initAgentDepths(NCU, CU) *//U is for 'unknown'*
90: known = 0;
91: **while** |NCU| + |CU| > reported + 1
92: receive $AgentDepth(x_i, NC(x_i) \setminus x, C(x_i), X)$, $x_i \in NCU$; known = known + 1;
93: **or** (receive $AgentDepth(x_i, NC(x_i), C(x_i) \setminus C_j, X)$, $x_i \in C_j \in$ CU;
94: **if** $\forall x_i$ from C_j have reported, known = known + 1;
95: **end while;**
96: unrep = $x_i \in$ NCU with AgentDepth not reported,
97: **or** $C_j \in$ CU with not all AgentDepths reported in the previous loop;
98: send AgentDepth(x,NC \ unrep, C \ unrep, X) to unrep;
99: receive AgentDepth(x_i, NC_i \ unrep, C_i \ unrep, X) from $\forall x_i \in$ unrep;
100: send AgentDepth(x, NC \ x_i, C \ x_i, X) to all $x_i \in$ (NC $\cup$ C) \ unrep;
101: send AgentDepth(x, X) to $\forall x_i \in$ (NC $\cup$ C);
102: receive AgentDepth(x_i, X) from $\forall x_i \in$ (NC $\cup$ C);
end initAgentDepths;

function orderUsingDFS(C_i, list)
110: append x to list; append BlockChildren(list[1]) to children and children_routes;
111: send list to BlockChildren(list[1]) $\cup$ ($\cup_{C_j \in C \setminus C_i} child(C_j)$) $\cup$ NC;
112: **for** $\forall x_i \in NC \cup C \setminus C_i$; *//get remaining children information*
113: receive "child y_i"; append y_i to children; append x_i to children_routes;
end for; end orderUsingDFS;

Figure 5: Limited block algorithm subroutines

function updateDepthsAndStructure(C_i, list)
120: send list to BlockChildren(list[last]) $\cup$ ($\cup_{C_j \in C \setminus C_i} child(C_j)$) $\cup$ NC;
121: remove C_i from C;
122: [newC, newNC] = rebuildBlock(C_i\list[last]);
123: add newC to C; add newNC to NC;
124: initAgentDepths(newNC, newC);
end updateDepthsAndStructure;

function updateDepths(list, y)
130: **if** $\exists C_j \in C$: y $\in C_j$
131: send list to BlockChildren(y) $\cup$ ($\cup_{C_j \in C: y \notin C_j} child(C_j)$) $\cup$ NC;
132: initAgentDepths($\emptyset$, C_j);
133: **else**
134: send list to ($\cup_{C_i \in C} child(C_i)$) $\cup$ (NC \ y);
135: initAgentDepths(y, $\emptyset$);
end if; end updateDepths;

Figure 6: Limited block algorithm subroutines

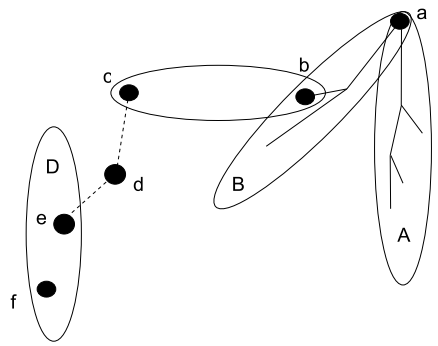

Figure 7: Limited-block example. Ellipses denote blocks, solid circles - agents, solid lines - ordering relationships, dotted lines - edges of the constraint graph

Definition 9 *For any block C in graph X*

$$BlockDepth(C, X) \equiv \min_{x \in C} AgentDepth(x, X) \tag{20}$$

Definition 10 *For constraint graph X*

$$GraphDepth(X) \equiv \min_{x \in X} AgentDepth(x, X) \tag{21}$$

The algorithm proceeds as follows. First, like in the acyclic case, the agents exchange messages to compute their own and their neighbors' *AgentDepth*s (60:). After this step is complete, the agent with the smallest depth value becomes the root for the current connected subgraph. Depending on their position relative to the root, other agents take different actions. Fig. 7 presents an example of the possible situations an agent can find itself in. Suppose agent a was chosen to be the root (62:-63:). To announce that it sends a message to its parent in the ordering (80:), and messages to its neighbors that are not in the same block, and to DFS children in the blocks (81:). To propagate this information for all the subgraphs, other agents forward these messages as necessary (65:, 111:, 120:, 131:, 134:). After sending its messages, a waits for the information about the children that it does not know about yet (83:-86:) and then exits.

Agents in the same block as root (68:) are ordered according to the DFS ordering of that block (69:, 82:, 110:-113:). For example, agent b is in the same block as a and gets its ordering according to the DFS spanning tree for that block rooted at a. Solid lines in Fig. 7 denote that clusters A and B become ordered according to DFS trees rooted at a.

Agents not in the same block as root, but in the same block as some other agent that becomes ordered (such as c that is in the same block as b, but not in the same block as a) (70:-71:) need to rediscover their block configuration, because after removal of a node the block may split in several blocks or even disappear alogether. The rediscovery is performed by the same function as the

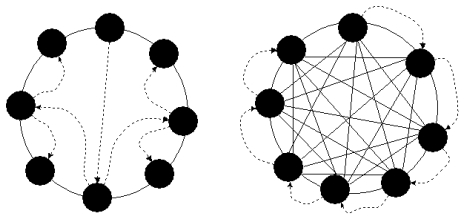

Figure 8: Simple-cycle and fully connected blocks

initial block information discovery, but restricted to agents of one block (122:-123:). For example, c restricts its effort to the agents in cluster C and does not attempt a new DFS traversal in the direction of node d. This is achieved by using $neighbors \cap (C_i \backslash \text{list[last]})$ instead of just $neighbors$ on line (46:) of the block discovery procedure.

The rest of the agents, those not in the same blocks with the newly-ordered nodes (72:-73:) (e.g. d, e, f) only need to forward the information about the new ancestors (131:, 134:) and update the relevant part of their $AgentDepth$s (132:, 135:). Then all the remaining agents participate in the next iteration of the ordering process.

An alternative to ordering the blocks that contain the root in a DFS manner would be to remove only root from consideration, rebuild the corresponding blocks and proceed to the next iteration. Either aproach can be used depending on how densely the nodes are connected inside blocks. If inside-block connectivity is sparse, the latter method is preferable, because it has a high chance of discovering a tree of smaller depth than any of the DFS spanning trees for this block. Otherwise the former method should be used, because it is faster and improvements over DFS are unlikely. Fig. 8 illustrates this idea with two extreme cases of inside-block connectivity: fully-connected block and simple-cycle block. From now on assume that the former approach (ordering whole clusters at once) is chosen.

3.4 Algorithm Properties

The following results show that for each iteration every agent can conclude whether it is a a root in a subgraph using only information about its neighbors and blocks it belongs to.

Theorem 6 *If X is a connected graph and there are 2 nodes $x_i, x_j \in X$ such that $AgentDepth(x_i, X) = AgentDepth(x_j, X) = depth(X)$, then x_i and x_j are either neighbors or belong to the same block.*

Proof: The proof by contradiction is similar to the acyclic case. Suppose $AgentDepth(x_1, X) = AgentDepth(x_n, X) = depth(X)$, and x_1 and x_n are not neighbors and do not belong to the same block. It means that there is a simple

path from x_1 to x_2, denote it x_1, x_2, ..., x_{n-1}, x_n, and at least one of the nodes on this path, x_i, separates x_1 and x_2. Suppose that x_i and x_{i-1} belong to the same block C_{i_1}, and x_i and x_{i+1} do not belong to the same block (other 3 possible cases have proofs that differ very little from this case).

By definition,

$$\begin{array}{l} AgentDepth(x_1, X) \geq dist(x_1, x_i, X) + AgentDepth(x_i, NC_i, C_i \setminus C_{i_1}, X) \\ AgentDepth(x_n, X) \geq dist(x_n, x_i, X) + AgentDepth(x_i, NC_i \setminus x_{i+1}, C_i, X) \end{array} \tag{22}$$

but then

$$\begin{array}{c} AgentDepth(x_i, X) = \\ \max\{AgentDepth(x_i, NC, C_i \setminus C_{i_1}, X), AgentDepth(x_i, NC_i \setminus x_{i+1}, C_i, X)\} < \\ < GraphDepth(X) \end{array} \tag{23}$$

which is a contradiction and shows that the assumption is wrong. □

Theorem 7 *If X is a connected graph, and for all neighbors $x_i \in NC$ and blocks $C_j \in C$ of a given node x holds*

$$AgentDepth(x, X) \leq AgentDepth(x_i, X) \tag{24}$$

$$AgentDepth(x, X) \leq BlockDepth(C_j, X) \tag{25}$$

then AgentDepth(x) = GraphDepth(X).

Proof: Again, the proof by contradiction is similar to the acyclic case. Suppose $AgentDepth(x_1, X) > AgentDepth(x_n, X) = depth(X)$, and

$$\begin{array}{ll} \forall x_i \in NC_1 & AgentDepth(x_1, X) \leq AgentDepth(x_{1_i}, X) \\ \forall C_{1_j} \in C_1 & AgentDepth(x_1, X) \leq BlockDepth(C_{1_j}, X) \end{array} \tag{26}$$

There is a simple path from x_1 to x_2, denote it x_1, x_2, ..., x_{n-1}, x_n, and at least one of the nodes on this path, separates x_1 and x_2. Denote x_i the first such node on the path from x_1 to x_n. It is either a neighbor of x_1 or belongs to the same block with x_1. Suppose that x_i and x_{i-1} belong to the same block C_{i_1}, and x_i and x_{i+1} do not belong to the same block (other 3 possible cases have proofs that differ very little from this case).

By definition,

$$\begin{array}{l} AgentDepth(x_1, X) \geq dist(x_1, x_i) + AgentDepth(x_i, NC_i, C_i \setminus C_{i_1}, X) \\ AgentDepth(x_n, X) \geq dist(x_n, x_i) + AgentDepth(x_i, NC_i \setminus x_{i+1}, C_i, X) \end{array} \tag{27}$$

but then

$$\begin{array}{c} AgentDepth(x_i, X) = \\ \max\{AgentDepth(x_i, NC_i, C_i \setminus C_{i_1}, X), AgentDepth(x_i, NC_i \setminus x_{i+1}, C_i, X)\} < \\ < max\{AgentDepth(x_1, X) - dist(x_1, x_i), AgentDepth(x_n, X) - dist(x_n, x_i)\} \end{array} \tag{28}$$

that is either $AgentDepth(x_i, X) < AgentDepth(x_1, X)$ (a contradiction with (26)), or $AgentDepth(x_i, X) < AgentDepth(x_n, X) = GraphDepth(X)$ (a contradiction with *GraphDepth* definition). □

Theorem 8 *If X is a connected graph, the result of the algorithm execution is a tree of depth no greater than $GraphDepth(X)$*

Proof: It is sufficient to prove that after removing a node x and its blocks from graph X for all the resulting connected subgraphs X_i

$$GraphDepth(X_i) \leq AgentDepth(x, X) - 1 \tag{29}$$

Because $\forall x_i \in NC \cap X_i$

$$\begin{gathered} AgentDepth(x_i, X_i) = \\ = AgentDepth(x_i, NC(x_i, X) \setminus x, C(x_i, X), X) \leq AgentDepth(x, X) - 1, \end{gathered} \tag{30}$$

and the theorem statement holds for all the subgraphs such that $X_i \cap NC \neq \emptyset$.

For subgraphs X_i that were connected to one of the removed blocks C_i, denote y_i the node separating X_i from x.

$$\begin{gathered} AgentDepth(y_i, X_i) \leq AgentDepth(x, X) - dist(x, x_i) \Rightarrow \\ \Rightarrow GraphDepth(X_i) < GrapthDepth(x, X) - dist(x, x_i) \end{gathered} \tag{31}$$

that concludes the proof. □

Theorem 9 *For any connected graph X the depth of the resulting tree is less than $\sqrt{2k|V|}$, where k is the maximum block size.*

The proof is analogous to the acyclic case, given the observation that after removal of a block from the graph there will remain a subgraph X_i such that $GraphDepth(X_i) \geq GraphDepth(X) - k$. □

3.5 Complexity

Estimate memory requirements. Parallel DFS search can require space up to $O(nk)$ (n threads with k-long depth). Block of size l requires $O(l^2)$ space, thus information about all the blocks takes up to $O(k^2\frac{n}{k}) = O(nk)$. Ancestors information $O(\sqrt{nk})$, and children information $O(n)$. Therefore, the worst-case memory complexity of the algorithm is $O(nk)$.

The time complexity of the algorithm (again, in synchronous cycles) is $O(n+|E|+k^{\frac{3}{2}}n^{\frac{1}{2}})$, where E is the number of edges in the constraint graph. The proof is as follows. The initial block discovery stage takes up to $O(n + |E|)$ time all the DFS traversals are executed simultaneously. The initial depth information propagation takes up to $O(GraphDepth(X)) = O(n)$. If x is the root of the tree, x_1 is its child, then it is enough to propagate the depth information for a distance of $dist(x, x_1) + k$ before x_1 can make a decision to become x's child. Block rediscovery also takes no more than $O(k)$ cycles per iteration. Because the maximum pseudo-tree depth is $O(\sqrt{kn})$, this adds the $O(k^{\frac{3}{2}}n^{\frac{1}{2}})$ term.

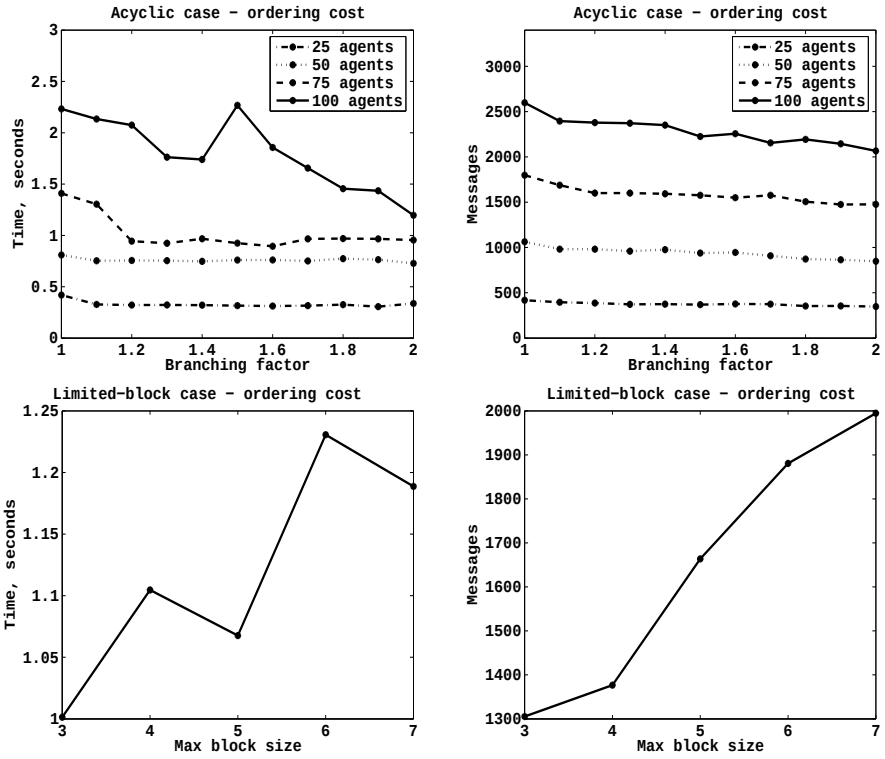

Figure 9: Ordering costs for acyclic (top row) and limited-block (bottom row) cases

4 Evaluation

The experimental evaluation consists of two parts. The first part is the computational and communicational cost of running the oredring algorithm itself. The second part shows how the ADOPT algorithm performs with pseudotree ordering compared to DFS ordering.

4.1 Ordering algorithm cost

The experimental measurements of the pseudo-tree ordering cost are summarized in Fig. 9. Limited-block case experiments were conducted for 50 agents, branching factor 1.7 and intra-cluster link density 0.5, varying only maximum cluster size. The algorithm was implemented using RETSINA [15] multiagent framework for message passing between agents. The time results were obtained on a single 3GHz Pentim 4 computer with 1GB of RAM. It was found that most of the computational resources was spent on parsing KQML messages.

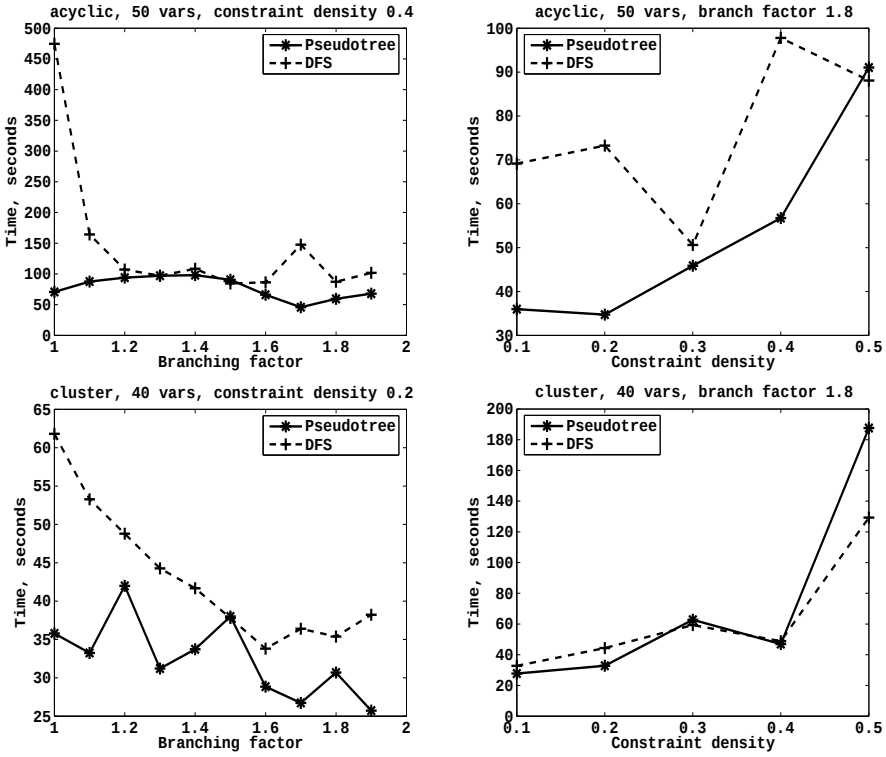

Figure 10: ADOPT performance using DFS and pseudotree orderings. Acyclic constraint graphs (top row) and limited-block case (bottom row)

4.2 ADOPT performance

We provide a comparison of the performance of ADOPT algorithm using DFS and pseudotree ordering of the variables (Fig. 10). We used only the algorithm runtime as a measure of performance, because communicational cost is linearly proportional to the runtime. Experiments with ADOPT were performed using agents running on 3 networked computers. Branching factor of the constraint tree (meta-tree in case of limited-block graphs) and average constraint tightness (fraction of the variables assignments that cause constraint violation) were varied. One can conclude that while neither ordering method dominates another for all types of problem structure, pseudotree ordering results in much faster DCOP solutions for problems with low constraint tightness.

5 Conclusion

We have presented a new algorithm for variable ordering, which allows to eliminate the need to process global information when solving a DCOP. It also in-

creases the solution search efficiency by exploiting the problem structure when possible. This extends the class of problems for which employing ADOPT is feasible and allows to apply it to completely decenralized problems. The algorithm has provable theoretical guarantee on the resulting ordering depth and polynomial time and space complexity.

6 Acknowledgements

This research was supported by AFOSR grant F49620-01-1-0542, AFRL/MNK grant F08630-03-1-0005, and NSF award IIS-0205526.

References

[1] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 16(1–2):149–180, 2005.

[2] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of International Joint Conference on Artificial Intellgence*, pages 1076–1078, 1985.

[3] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 222–236, 1997.

[4] Pragnesh Jay Modi, Syed Ali, Wei-Min Shen, and Milind Tambe. Distributed constraint reasoning under unreliable communication. In *Proceedings of Distributed Constraint Reasoning Workshop at International Joint Conference on Autonomous Agents and Multiagent Systems*, 2003.

[5] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems*, 2004.

[6] Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. Argumentation as distributed constraint satisfaction: applications and results. In *AGENTS '01: Proceedings of the Fifth International Conference on Autonomous Agents*, pages 324–331, 2001.

[7] Katsutoshi Hirayama, Makoto Yokoo, and Katia Sycara. An easy-hard-easy cost profile in distributed constraint satisfaction. *Information Processing Society of Japan Journal*, 45(9):2217–2225, 2004.

[8] Makoto Yokoo. *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer-Verlag, London, UK, 2001.

[9] Vipin Kumar. Algorithms for constraint-satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, 1992.

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

[11] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[12] Roberto J. Bayardo Jr. and Daniel P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Proceedings of International Joint Conference on Artificial Intellgence*, pages 558–562, 1995.

[13] Katsutoshi Hirayama and Makoto Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of International Conference on MultiAgent Systems*, pages 135–142, 2000.

[14] Reinhard Diestel. *Graph Theory*. Springer, New York, NY, USA, 1997.

[15] Katia Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1,2), 2003.