Multiply-Constrained Distributed Constraint Optimization

E. Bowring, M. Tambe
Computer Science Dept.
Univ. of Southern California
Los Angeles, CA 90089
{bowring,tambe}@usc.edu

M. Yokoo
Dept. of Intelligent Systems
Kyushu University
Fukuoka, 812-8581 Japan
yokoo@is.kyushu-u.ac.jp

ABSTRACT

Distributed constraint optimization (DCOP) has emerged as a useful technique for multiagent coordination. While previous DCOP work focuses on optimizing a single team objective, in many domains, agents must satisfy additional constraints on resources consumed locally (due to interactions within their local neighborhoods). Such resource constraints may be required to be private or shared for efficiency's sake. This paper provides a novel *multiply-constrained DCOP* algorithm for addressing these domains which is based on mutually-intervening search, i.e. using local resource constraints to intervene in the search for the optimal solution and vice versa. It is realized through three key ideas: (i) transforming n-ary constraints to maintain privacy; (ii) dynamically setting upper bounds on joint resource consumption with neighbors; and (iii) identifying if the local DCOP graph structure allows agents to compute exact resource bounds for additional efficiency. These ideas are implemented by modifying Adopt, one of the most efficient DCOP algorithms. Both detailed experimental results as well as proofs of correctness are presented.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Design, Theory

Keywords

constraint reasoning, DCOP, multiagent systems, privacy

1. INTRODUCTION

Distributed Constraint Optimization (DCOP) [1,6,9,11] is a useful technique for multiagent coordination with applications in distributed meeting scheduling and distributed factory and staff scheduling [8]. In a DCOP, cooperative distributed agents, each in control of a set of variables, assign values to these variables, so as to optimize a global objective function expressed as an aggregation of utility functions over combinations of assigned values.

While recent advances in efficient DCOP algorithms are encouraging [1, 6, 9, 11], these algorithms focus on optimizing a single objective and fail to capture the complexities that arise in many domains where agents must adhere to resource constraints, e.g. budgets or fuel. These resource constraints necessitate multiply-constrained DCOP algorithms that optimize a global objective, while ensuring that resource limits are not exceeded. While in some domains agent must keep these resource constraints private (e.g. travel budgets in distributed meeting scheduling [8]), in others resource constraints may be non-private (e.g. overtime limits in staff allocation for distributed software development).

There are three main challenges that must be addressed in designing multiply-constrained DCOP algorithms. First, agents' n-ary resource constraints add to DCOP search complexity. Hence, agents must quickly prune unproductive search paths. Second, harnessing state-of-the-art DCOP algorithms (henceforth, referred to as "singly-constrained" DCOP algorithms) is crucial. However, existing algorithms are not deisgned to handle resource constraints that are local, possibly private and defined over n-ary domains. Third, algorithms must exploit constraint revelation to gain efficiency when privacy is not required. (Unlike work which uses cryptographic techniques in DisCSP/DCOP for privacy [12, 15], we do not insist on such watertight privacy, given potentially significant communication and computational overheads.)

This paper presents a novel multiply-constrained DCOP algorithm which employs *mutually-intervening searches* to address the first challenge above: while an agent immediately intervenes in the search for the global optimal if its local resource constraint is violated, opportunistic search for the global optimal solution obviates testing all partial solutions for resource constraint satisfaction. There are three key ideas in this algorithm, which are facets of mutually-intervening search: constraint-graph transformation, dynamically-constraining search and local acyclicity. The first idea, *constraint-graph transformation*, is motivated by the second challenge above: harnessing singly-constrained DCOP algorithms for multiply-constrained DCOPs. The key innovation is in efficiently employing virtual variables to enforce an agent's resource constraints within the singly-constrained DCOP algorithms. These variables indicate resource constraint violations with asynchronous communications of high negative costs, which preemptively prune search paths. Yet, for correctness and privacy preservation, we restructure the DCOP graph and appropriately place the virtual variables in that graph. The following two ideas address the third challenge above by exploiting the privacy-efficiency tradeoff. In *dynamically-constraining search*, an agent reveals to its neighbors an upper-bound on any non-private resource constraint. When optimizing the global objective function, its neighbors only select values that abide by the bounds, improving algorithmic efficiency. The final idea, *local acyclicity*, further improves efficiency in locally acyclic DCOP graphs by allowing the communicated resource bounds to be tightened without sacrificing algorithmic correctness. In particular, we define T-nodes, which are variables whose local graph acyclicity allows for the dynamic use of exact bounds and do not require virtual variables. We show that these tighter bounds can not be applied at non-T-nodes.

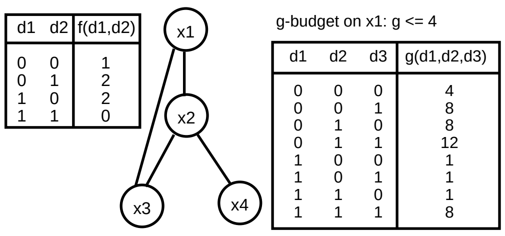
While we illustrate these ideas by building on top of Adopt, one of the most efficient DCOP algorithms [9], our techniques could be applied to other algorithms, e.g. OptAPO, SynchBB [6, 14]. We present a multiply-constrained Adopt algorithm that tailors its performance to three situations: 1) when a resource constraint must be kept private, 2) when a constraint is sharable but the variable is not a T-node, and 3) when a constraint is sharable and the variable is a T-node. These different techniques can be employed simultaneously by different variables in the same problem. We experimentally compare these techniques and illustrate problems where agents gain the most efficiency by sharing resource constraints and problems where agents lose no efficiency by maintaining privacy.

2. PROBLEM DEFINITION

2.1 DCOP

A DCOP [1, 9, 11] consists of n variables, $\{x_1, x_2, \ldots, x_n\}$, assigned to a set of agents who control the values they take on. Variable x_i can take on any value from the discrete finite domain D_i. The goal is to choose values for the variables such that the sum over the set of constraint cost functions (f_{ij}) is minimized, i.e. find an assignment, A, s.t. F(A) is minimized: $F(A) = \sum_{x_i,x_j \in V} f_{ij}(d_i, d_j)$, where $x_i \leftarrow d_i, x_j \leftarrow d_j \in A$. Variables x_i and x_j are considered neighbors since they share a constraint.

In the example in Figure 1, x_1, x_2, x_3, and x_4 are variables each with domain {0,1} and identical f-cost functions on each link (shown for the x_1 to x_2 link), $F(x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0, x_4 \leftarrow 0) = 4$ and the optimal assignment would be $(x_1 \leftarrow 1, x_2 \leftarrow 1, x_3 \leftarrow 1, x_4 \leftarrow 1)$. While the above commonly used definition of DCOP emphasizes binary constraints, *general DCOP representations* may include n-ary constraints.



d1	d2	f(d1,d2)
0	0	1
0	1	2
1	0	2
1	1	0

d1	d2	d3	g(d1,d2,d3)
0	0	0	4
0	0	1	8
0	1	0	8
0	1	1	12
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	8

Figure 1: Multiply-Constrained DCOP graph

2.2 Multiply-Constrained DCOP

In Multiply-Constrained DCOP we make use of n-ary DCOP constraints by adding a new cost function g_i on a subset of x_i's links and a g-budget G_i which the accumulated g-cost must not exceed. Together this g-function and g-budget constitute a g-constraint. Figure 1 shows an example g-constraint on x_1. In the example, if x_1, x_2, x_3 each take on the value of 1 (leading to an optimal f-cost) then the g-cost is 8, which violates x_1's g-budget of 4. Since g-cost functions cannot be merged with f-cost functions, each value must be selected based on both f and g, and hence this is a multiply-constrained DCOP. It is straightforward to extend this framework to multiple g-constraints on a variable.

In general, the combined g-cost can be an arbitrary function on the values of x_i and its neighbors.[1] We define g-constraints to be private or shared. If a g-constraint can be shared and the g-cost function is the sum of the g-costs of the links impinging upon x_i, we can improve efficiency by exploiting the additive nature of the function. In the rest of the paper, we assume such additivity for shared g-constraints but make no such assumption for private g-constraints. Given the g-cost functions and g-budgets, we now modify the DCOP objective to be: find A s.t. F(A) is minimized: $F(A) = \sum_{x_i,x_j \in V} f_{ij}(d_i, d_j)$, where $x_i \leftarrow d_i, x_j \leftarrow d_j \in A$
and $\forall x_i \in V$

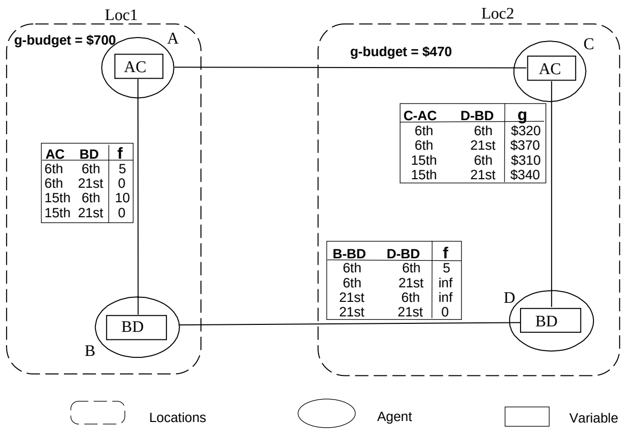
$g_i(d_i, \{d_j | x_j \in neighbors(x_i)\}) \leq G_i$

Multiply-Constrained DCOP is situated within the space of general DCOP representations mentioned in Section 2.1. However, it emphasizes three features: (i) n-ary (g-cost) constraints, (ii) private g-constraints, and (iii) the need to exploit the interaction between f- and g-constraints for pruning the search space. No current DCOP algorithm, including leading algorithms, is able to address all three issues. Specifically, Adopt [9] can handle (ii) but not (i) or (iii). OptAPO [6] and DPOP [11] handle (i) but not (ii) or (iii).

2.3 Motivating Domains

In this section we demonstrate the need for multiply-constrained DCOP algorithms with two example domains. These domains require agents to optimize an f-cost function and yet adhere to (private) g-constraints.

Distributed Meeting Scheduling: When members of organizations in separate locations collaborate, personal assistant agents must optimize their meeting schedules and yet adhere to travel budgets. Consider the example in Figure 2, where researchers, A, B, C and D are divided between organizations at Loc1 and Loc2. A and C need to meet, and B and D also need to meet. A and C are group leaders who manage travel budgets for their groups and wish to keep budgetary constraints private.



AC	BD	f
6th	6th	5
6th	21st	0
15th	6th	10
15th	21st	0

C-AC	D-BD	g
6th	6th	$320
6th	21st	$370
15th	6th	$310
15th	21st	$340

B-BD	D-BD	f
6th	6th	5
6th	21st	inf
21st	6th	inf
21st	21st	0

Figure 2: Meeting Scheduling Example

Multiply-constrained DCOP allows us to model this problem. We assume that agents share all their scheduling preferences (the exact representation of these preferences is not important for this example). The domain of each variable is a tuple of time-of-meeting

[1]While for simplicity we assume that the set of f-neighbors and g-neighbors are the same, this is not necessary.

and location, e.g. {6th Oct at Loc1, 15th Oct at Loc2}. The f-cost expresses agents' scheduling preferences and that they must agree on the meeting time (or else receive an infinite cost). The f-cost function between C and D reflects C's preference that the BD meeting precede the AC meeting (for readability not all of the f- and g-functions are shown in Figure 2). Since a meeting may require flying to that destination, the g-cost represents travel costs. The cost varies depending on the date of travel and whether all members are traveling on the same day and can share a cab. C has a travel budget of \$470, represented via a g-budget on C's variable AC. C must pay for expenses incurred by C or D in either the AC or BD meeting if they are held at Loc1. A and C may not wish to reveal their budgets (hence private g-constraints).

Distributed Software Development: Many software companies have campuses across the globe and teams collaborate across both distances and time zones [3,5]. While interdependent tasks within a project must be scheduled for their timely completion, a team liaison must videoconference — often after normal workhours — with the team to which the code is being sent to facilitate the handoff. Figure 2 shows an example involving five tasks $\{t_1, \ldots, t_5\}$, where agents in Team1 must complete t_1 and t_4 and Team2 must complete t_2, t_3 and t_5. The domain values are times at which a task can be completed. The f-function captures both the temporal precedence constraints and a preference function over the potential times (e.g. t_1 must complete before t_2, and prefer t_1 to start early in the day). A team liaison from Team1 must videoconference with Team2 during the first hour of t_2 and t_3, requiring the liaison to work overtime. To avoid burnout corporate policy may limit liaison overtime to 8 hours, establishing a non-private g-constraint.

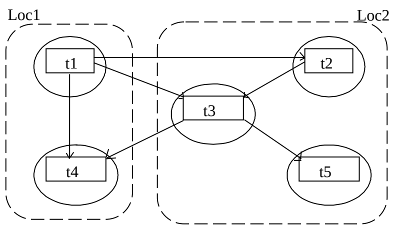


Figure 3: Software Development

3. BACKGROUND: ADOPT

Adopt [9] is an asynchronous complete DCOP algorithm. Adopt has been discussed in detail in the literature [1,2,9], so we provide only a brief description. Adopt organizes variables into a Depth-First Search (DFS) tree in which constraints are allowed between a variable and its ancestors or descendants, but not between variables in separate sub-trees. The constraint graph in Figure 1 is organized as a DFS tree. x_2 is a child of x_1 and x_3 is a descendant (but not a child) of x_1. In this paper, we will use the terms variable and node interchangeably.

Adopt employs two basic messages: VALUE and COST.[2] Assignments of values to variables are conveyed in VALUE messages that are sent to variables lower in the DFS tree, that share a constraint with the sender. For example, x_1 will send its VALUE messages to x_2 and x_3. To start, variables take on a random value and send out VALUE messages to get the flow of computation started. A COST message is sent from children to parents indicating the f-cost of the sub-tree rooted at the child (e.g. x_3 will send its COST messages to x_2 and x_2 sends COST messages to x_1). A variable keeps its current assignment until the lower bound on cost accumulated, i.e. the lower bound of its children's sub-trees plus the f-cost of its constraints with its ancestors, exceeds the lower-bound cost of another assignment. When this occurs, the variable will *opportunistically* switch its value assignment (unexplored assignments have a lower bound of zero). The root's upper and lower bounds represent the upper and lower bounds on the global problem; when they meet the optimal has been found and the algorithm terminates. Since communication is asynchronous, messages have a context, i.e. a list of the variable assignments in existence at the time of sending, attached to them to help determine information relevance.

[2]Adopt also uses THRESHOLD messages for improved efficiency, but this is orthogonal to the contributions in this paper.

4. MULTIPLY-CONSTRAINED ADOPT

This section is organized as follows: Section 4.1 describes the key ideas in Multiply-Constrained Adopt (MCA), Section 4.2 describes MCA in detail, and Section 4.3 provides correctness and complexity results.

4.1 Basic Ideas

The mutually intervening approach of MCA is realized through four techniques: constraint-graph transformation, dynamically constraining search, local acyclicity and exploiting local independence.

Constraint-graph transformation To exploit existing DCOP algorithms and maintain privacy, we add virtual variables to enforce n-ary g-constraints. The virtual variables asynchronously communicate infinite costs on g-constraint violations, preemptively pruning search paths, e.g. in Figure 4a, x_1' is added to represent a constraint between x_1, x_2, x_3. While using extra variables to enforce n-ary constraints using binary ones has appeared in the centralized CSP literature [13], the novelty of our contribution is threefold. First, we embed these virtual variables in a singly-constrained DCOP algorithm and prove the correctness of the resulting asynchronous multiply-constrained DCOP algorithm. To that end, we restructure Adopt's DFS tree and appropriately place the virtual variable in the tree. Second, since such constraint-graph restructuring can change local acyclicity properties of other nodes, the preprocessing identifies and preserves local acyclicity properties where feasible. Third, by encapsulating an n-ary constraint in a virtual variable owned by the original variable's owner and placing it as a leaf in the DFS tree, we can protect the privacy of both the g-function and the g-budget.

Dynamically Constraining Search When permissible, it is important to exploit g-constraint revelation to gain efficiency in the f-optimization. We achieve this by requiring descendant nodes to only consider assignments that will not violate their ancestors' g-constraints. Specifically, we pass each descendant a bound (termed *g-threshold*) specifying how large a g-cost it can pass up, limiting its domain. This g-threshold represents an exact bound when the local graph is acyclic and an upper bound otherwise. If a value fails to satisfy an ancestor's g-threshold, nodes will not explore this value for f-optimality. Additionally, the opportunistic search for an optimal f necessitates checking for g-constraint violations only for those value combinations that are of low f-cost. Thus, the searches dynamically constrain each other, leading to performance improvements.

Local Acyclicity (T-nodes) The notion of local acyclicity is captured formally in our definition of T-nodes: variable x_i is a T-node if all neighbors of x_i lower in the DFS tree are children of x_i. In Figure 1, x_1 is not a T-node because x_3 is not its child, but x_2, x_3 and x_4 are all T-nodes. T-nodes enable the calculation of exact g-thresholds and elimination of virtual variables because their children respond independently to allocated g-thresholds.

```
% In preprocessing we assume an initial DFS tree has been built.
% Convention: $x_i$ is self, $x_j$ is a higher priority neighbor
% and $x_k$ and $x_l$ are lower priority neighbors.
Preprocessing
(1)  **for** each $x_i$ from highest priority to lowest
(2)      **if** $Tnode_i == false$ or $private_i == true$
(3)          $x'_i$ is a new virtual variable
(4)          $Neighbors(x'_i) \leftarrow Neighbors(x_i) \cup x_i$
(5)          $Neighbors(x_i) \leftarrow Neighbors(x_i) \cup x'_i$
(6)          **forall** $x_k \in Children(x_i)$
(7)              **if** $x_k$ is not a neighbor of $x_l \in Children(x_i)$
(8)              $Neighbors(x_k) \leftarrow Neighbors(x_k) \cup x_l$
(9)          rebuildDFStree($x_1 \ldots x_n$)
(10) **forall** virtual variables $x'_i$,
(11)     $parent(x'_i) \leftarrow$ lowest priority Neighbor of $x'_i$

Initialize
(12) $CurrentContext \leftarrow \{\}$
(13) initialize structures to store lb and ub from children
(14) $d_i \leftarrow d$ that minimizes $LB(d)$
(15) **if** $private_i == false$ and $Tnode_i == true$
(16)     **forall** $x_l \in Children$
(17)         **for** $gt \leftarrow 0 \ldots G_i$
(18)             $GFmap(x_l, gt) \leftarrow min\ f(d_i, d_l)$ s.t. $g(d_i, d_l) \leq gt$
(19) **if** $Tnode_i == true$ and $private_i == false$
(20)     calcOptimalSplit
(21) **else if** $private_i == false$
(22)     calcUpperBound
(23) backTrack

when received (**VALUE**, $x_j$, $d_j$, $gThresh_j$)
(24) **if** $private_j == false$
(25)     add ($x_j$,$d_j$,$gThresh_j$) to $CurrentContext$
(26) **else**
(27)     add ($x_j$,$d_j$) to $CurrentContext$
(28) reset lb and ub if $CurrentContext$ has changed
(29) **if** $private_i == false$ and $Tnode_i == true$
(31)     **if** $CurrentContext$ has changed
(30)         **forall** $x_l \in Children$
(32)             $GFmap(x_l, gt) \leftarrow min\ f(d_l, d_i)$ s.t. $g(d_l, d_i) \leq gt$
(33)     calcOptimalSplit
(34) backTrack;

when received (**COST**, $x_k$, $context$, $lb$, $ub$)
(35) update $CurrentContext$
(36) **if** $context$ compatible with $CurrentContext$
         and $Tnode_i == true$ and $private_i == false$
(37)     $GFmap(x_l, gThresh_{il}) \leftarrow lb$ s.t.
             $(x_i, d_i, gThresh_{il})$ is part of $context$ from $x_k$
(38)     calcOptimalSplit
(39)     **if** any $gThresh_{il}$ has changed, reset lb,ub
(40)     **else** store lb,ub
(41) **else if** $context$ compatible with $CurrentContext$ and $gThresh_{il}$
(42)     store lb and ub
(43) backTrack

procedure backTrack
(44) **if** $x_i$ not a virtual variable
(45)     **if** no $d$ satisfies $gThresh_j$ $LB, UB \leftarrow \infty$
(46)     **else if** $LB(d_i) > LB(d)$ for some $d$
(47)         $d_i \leftarrow d$ that minimizes $LB(d)$ and satisfies $gThresh_j$
(48)         **if** $Tnode_i == true$ and $private_i == false$
(49)             $GFmap(x_l, gt) \leftarrow min\ f(d_i, d_l)$ s.t. $g(d_i, d_l) \leq gt$
(50)             calcOptimalSplit; reset lb and ub
(51)         **if** $private_i == false$
(52)             SEND (**VALUE**, ($x_i$, $d_i$, $gThresh(x_k)$))
                 to each lower priority neighbor $x_k$
(53)         **else** SEND (**VALUE**, ($x_i$, $d_i$))
                 to each lower priority neighbor $x_k$
(54)     **if** $LB == UB$:
(55)         **if** TERMINATE received from parent or $x_i$ is root:
(56)             SEND TERMINATE to each child
(57)             Terminate execution;
(58)     SEND (**COST**, $x_i$, $CurrentContext$, $LB$, $UB$)
(59) **else** % else a virtual variable
(60)     **if** $g(d_i, CurrentContext) > gBudget(x_i)$
(61)             SEND (**COST**, $x_i$, $CurrentContext$, $\infty$, $\infty$) to parent
(62)     **else**
(63)         SEND (**COST**, $x_i$, $CurrentContext$, 0, 0) to parent
```

Figure 4: Multiply-Constrained Adopt Pseudo-code

Exploiting Local Independence In applying different techniques to different nodes in a single problem, we exploit the locality of the g-constraints.

4.2 MCA Algorithm Description

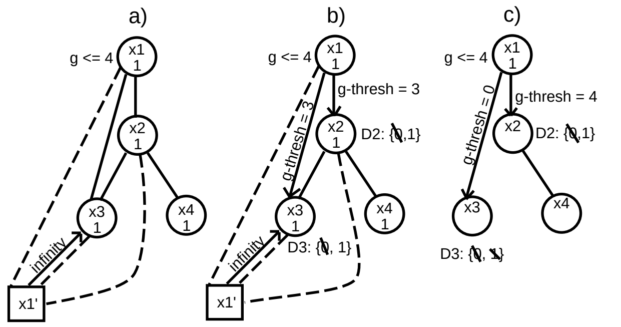


Figure 5: a) MCAP b) MCAS c) MCASA

Figure 4 presents pseudo-code for MCA. To emphasize new elements we include only brief descriptions of functionality unchanged from Adopt [9]. Lb and ub refer to the lower and upper bounds on children's f-costs for each value d_i while LB and UB refer to the overall lower and upper bound at x_i. MCA uses three subalgorithms: (i) MCA Private (MCAP) is used when a constraint may not be revealed, and it uses constraint-graph transformation, (ii) MCA Shared (MCAS) is used when a constraint is non-private but the variable is not a T-node, and it employs constraint-graph transformation and dynamically constraining search, and (iii) MCA Shared and Acyclic (MCASA) is utilized when a constraint is non-private and the variable is a T-node, and it uses dynamically constraining search. We take privacy to mean that neither the g-constraints nor the variables involved are explicitly revealed to other agents. We will discuss each technique separately and then indicate how they interact in the unified algorithm.

4.2.1 MCAP

An example snapshot of MCAP's execution is shown in Figure 5a. MCAP searches for an optimal solution for f and when an assignment violates a g-constraint, a feedback signal of infinite cost is sent to the cluster of nodes involved, zero cost otherwise (lines 60-63 in Figure 4). The feedback is sent by a virtual variable (x'_1) which is responsible for enforcing the g-constraint of a single variable (x_1); the virtual variable is controlled by the agent that owns the original variable (lines 3-5). The feedback is sent to the lowest priority variable involved in the violated constraint (x_3). Using the f-cost optimization mechanisms of Adopt, the feedback will propagate to the node(s) that must change their values to find the optimal satisfying solution. Specifically, if a variable (e.g. x_3) receives an infinite cost for all of its domain values, it will send a COST message to its parent (e.g. x_2) indicating an lb and ub of ∞ which will prompt the parent to change its value. Since feedback must be able to percolate up to all the nodes in a constraint, Adopt's DFS tree must be restructured to put all the variables in a g-constraint in the same subtree (lines 6-8). MCA preprocessing creates the virtual variables and places them as leaves, lower in the DFS tree than the variables in the g-constraint they enforce (lines 10-11). We revisit preprocessing of DFS tree in Section 4.2.4.

4.2.2 MCAS

While feedback signals maintain the privacy of the g-constraints and the variables involved by encapsulating the g-constraint inside a variable, MCAP's partial search of unsatisfying assignments slows the algorithm. When a g-constraint is non-private, we reveal the g-function of a link to those nodes connected to it and use this information to implement MCAS and MCASA shown in Figures 5b and 5c respectively.

MCAS and MCASA exploit g-function revelation by having nodes send their descendants g-thresholds (line 52) indicating that the descendant may take on no value with a g-cost greater than the g-threshold. In the snapshots from Figures 5b and 5c we can see the g-thresholds being passed down from x_1 to x_2 and x_3 (as discussed earlier, we assume that the g-functions are additive here). If the variable is not a T-node (in MCAS), the g-thresholds for a child (x_l) are an upper bound: the total g-budget minus the minimum g-cost that could be consumed by each of the other links ($G_i - \sum_{x_j \in Neighbors(x_i) \neq x_l} min\ g_{ij}(d_i, d_j)$). In Figure 5b, the total g-budget at x_1 is 4, and each link consumes at least 1, so the g-thresholds are 3. Given this bound, a node can prune values from its domain (e.g. 0 is pruned from the domain of x_3) leading to speedups over MCAP.

4.2.3 MCASA

For T-nodes, it is possible to calculate an exact bound (MCASA) enabling more values to be pruned out and further speeding up the search (e.g. x_1 in 5c now a T-node, calculates exact bounds). Calculating the exact bounds requires a node to maintain a mapping from potential g-thresholds to lower bounds on f-costs (GFmap) for each of its children. GFmap is dynamically initialized for the current value from the link function (line 17-18; line 50) and then updated as COST messages arrive (line 38). A T-node uses these g-threshold to f-cost mappings to calculate how to split its remaining g-budget (its initial g-budget minus the g-cost consumed on each of the links with its higher priority neighbors) among its children. This calcOptimalSplit function can be implemented using a straightforward dynamic program and thus is omitted from the pseudo-code. Since x_i's lb and ub for the current context now depend upon the way x_i splits its g-budget among its children, we store lb and ub based upon x_i's current split. We reset lb and ub to 0 and ∞ respectively whenever x_i changes its split (line 39) as well as when the context changes (line 28). If a node can't satisfy any of its g-thresholds, it will send up a cost of ∞ which will cause its ancestors to switch their values (line 45).

4.2.4 Combining Techniques

Until now we have discussed each technique separately. However, the techniques may be applied simultaneously to different nodes within the same problem. Since the g-constraints are local constraints, we only need worry about situations where a parent and a child are using different techniques: (i) Parent = MCAS/MCASA, Child = MCAP: The child can restrict its domain based upon the g-threshold of its parent and the parent will automatically change its value if the child's virtual variable passes up an infinite cost. (ii) Parent = MCAP, Child = MCAS/MCASA: The child will automatically adjust its value if the parent's virtual variable passes up an infinite cost and the child's technique makes no difference to the parent.

One key issue is whether the tree transformations can turn a T-node somewhere else in the tree into a non-T-node. Two steps can cause links to be added: (i) adding empty links between x_i's children to force them into the same subtree (lines 6-8) and (ii) adding a virtual variable and empty links to each of the variables in the g-constraint it represents (lines 10-11). If a link is added between two children of a T-node, then despite having no f-function or g-function it can turn a T-node into a non-T-node. For instance, in

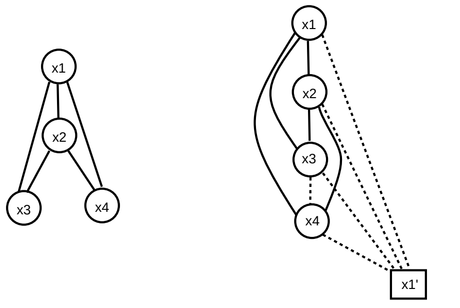


Figure 6: a) original constraint graph b) after adding x_1'

Figure 6, x_1 has a g-constraint with x_2, x_3 and x_4. During preprocessing, a link must be added between x_3 and x_4 to keep them from being in separate subtrees and one must be chosen (arbitrarily) to be the parent of the other (x_3 is made parent of x_4). x_4's COST messages no longer flow directly to x_2 but are sent via x_3 where they are combined with x_3's local costs. This combination prevents x_2 from calculating an exact bound and renders it no longer a T-node.

In contrast, in Figure 4a the addition of virtual variables and their accompanying links, does not change a T-node to a non-T-node. While x_2 is originally a T-node, the addition of the virtual variable x_1' creates a lower priority neighbor for x_2 that is not a child. However, in this case, x_1' is a virtual variable, i.e. it has no f-functions or g-functions on any of its links and it has no domain of its own. Since x_1' need be given no g-threshold, the fact that it is not one of x_2's children does not prevent x_2 from being a T-node.

With these two cases in mind, preprocessing (lines 1-11) starts by walking through the tree in priority order adding in the empty links between children where necessary and adjusting the priorities accordingly. Once a node, x_i, has been determined to be a T-node no links added between its lower priority non-neighbors will change x_i into a non-T-node, so one sweep of the tree is sufficient to correctly determine which nodes are T-nodes. After this sweep the virtual variables themselves can be added as leaves without causing any of the previously determined T-nodes to lose that property.

4.3 Correctness and Complexity of MCA

In this section we will again separate out the proofs for each technique for the sake of clarity. As previously described, the interaction of the techniques in the combined algorithm does not change their properties. Recall in the following that a context is the set of variable assignments upon which a piece of information is predicated.

PROPOSITION 1. *For each node x_i for the current context, MCAP finds the assignment whose f-cost, local cost (δ) plus the sum of each of x_i's children's (x_l's) costs, is minimized while satisfying the g-constraint:*

$$OPT(x_i, context) \overset{def}{=} min_{d \in D_i}[\delta(d) + \sum_{x_l} OPT(x_l, context \cup (x_i, d))] \quad if\ g_i(d_i, \{d_j | x_j \in Neighbors(x_i)\}) \leq G_i; \quad \infty\ otherwise$$

$$where\ \delta(d) \overset{def}{=} \sum_{x_j \in ancestors} f_{ij}(d_i, d_j)$$

Proof: To show this we start from the correctness of the original Adopt algorithm [9]. Adopt will find at every node x_i:

$$OPT'(x_i, context) \overset{def}{=} min_{d \in D_i}[\delta'(d) + \sum_{x_l} OPT'(x_l, context \cup (x_i, d))]$$

$$\text{where } \delta'(d) \overset{def}{=} \sum_{x_j \in ancestors} f_{ij}(d_i, d_j)$$

To show that MCAP finds $OPT(x_i, context)$ we show that 1) it never returns an assignment containing a violated g-constraint, unless the g-constraint is unsatisfiable and 2) it finds the minimum f-cost solution.

The proof of part 1) starts with the fact that the virtual variables introduce an infinite f-cost into the subtree containing the violated constraint. This infinite f-cost enters lower in the priority tree than any variable in the constraint which allows the normal flow of COST messages to eventually carry it to all of the involved nodes. Since any assignment that does not violate the g-constraint will have a finite f-cost, it follows from the correctness of Adopt that by choosing the assignment that minimizes f, MCAP will never take on an assignment that violates a g-constraint unless the g-constraint is unsatisfiable. Part 2) follows directly from the correctness of Adopt because the virtual variables report a zero cost if all constraints are satisfied, which means that by Adopt's normal mechanisms it will find the minimum f-cost solution. ■

Proving that MCAS is correct requires a minor addition to the MCAP proof from Proposition 1 which is shown below.

PROPOSITION 2. *If the g-constraint for each node x_i is $\sum_{x_j \in Neighbors(x_i)} g_{ij}(d_i, d_j) < G_i$, then no satisfying solution can contain on link l_{il} a g-cost greater than $G_i - \sum_{x_j \in Neighbors(x_i) \neq x_l} min\ g_{ij}(d_i, d_j)$.*

Proof: Each link consumes a certain minimum g-cost, and we are only subtracting the sum of the absolute minimum costs on all links. ■

We next turn to MCASA; if we can prove the g-thresholds are assigned optimally, then taken with original Adopt's correctness it will show that MCASA is correct. After that we will show that when a node is not a T-node then exact bounds do not apply.

PROPOSITION 3. *For each T-node x_i, MCASA always terminates with an optimal division of the g-budget given d_i and the current context.*

Proof by Contradiction: Throughtout the proof we assume $x_k \in Children(x_i)$ and we assume there are no T-nodes as descendents of x_i – using induction we can generalize. Assume MCASA terminates with g-thresholds g'_{ik} ($\forall x_k$) which are not optimal. Thus there exists another set of g-thresholds (g^*_{ik}) s.t. $\delta(d_i) + \sum_{x_k} min\ lb(d_i, x_k)$ where $d_k \in \{D_k | g_{ik}(d_i, d_k) \leq g_{ik}*\} < \delta(d_i) + \sum_{x_k} min\ lb(d_i, x_k)$ where $d_k \in \{D_k | g_{ik}(d_i, d_k) \leq g'_{ik}\}$. Since local cost $\delta(d)$ is constant for all g-thresholds, we will drop it. To have been selected, g'_{ik} must have seemed optimal based on current information at some point. To distinguish these two states of knowledge we coin the following terms: $f_{actual}(g_{ik}, x_k)$ is the f-cost (specifically $min\ lb(d_i, x_k)$ where $d_k \in \{D_k | g_{ik}(d_i, d_k) \leq g_{ik}\}$) when all costs have percolated up from all descendants and $f_{current}(g_{ik}, x_k)$ is the current lower bound on f-cost. When g'_{ik} is selected:

1. $\sum_{x_k} f_{actual}(g'_{ik}, x_k) > \sum_{x_k} f_{actual}(g^*_{ik}, x_k)$
2. $\sum_{x_k} f_{current}(g'_{ik}, x_k) < \sum_{x_k} f_{current}(g^*_{ik}, x_k)$
3. $f_{current}(g_{ik}, x_k) \leq f_{actual}(g_{ik}, x_k)$ for any g_{ik}
4. $\sum_{x_k} f_{current}(g_{ik}, x_k) \leq \sum_{x_k} f_{actual}(g_{ik}, x_k)$

Since there are a finite number of nodes in the tree below x_i, before termination $\sum_{x_k} f_{current}(g'_{ik}, x_k) = \sum_{x_k} f_{actual}(g'_{ik}, x_k)$ will become true. At this point:

$$\sum_{x_k} f_{current}(g'_{ik}, x_k) \quad = \quad \sum_{x_k} f_{actual}(g'_{ik}, x_k)$$

$$\sum_{x_k} f_{current}(g'_{ik}, x_k) \quad > \quad \sum_{x_k} f_{actual}(g^*_{ik}, x_k) \qquad \text{by (1)}$$

$$\sum_{x_k} f_{current}(g'_{ik}, x_k) \quad > \quad \sum_{x_k} f_{current}(g^*_{ik}, x_k) \qquad \text{by (4)}$$

Thus based upon current information ($f_{current}$) MCASA will switch from g'_{ik} to $g_{ik}*$ because it has a lower associated f-cost. This contradicts our assumption that MCASA will terminate with g-thresholds g'_{ik}. ■

If x_i is not a T-node, then MCASA is not guaranteed to find the assignment whose f-cost, local cost (δ) plus the sum of x_i's children's costs, is minimized while satisfying the g-constraint. We can show this with a counter-example.

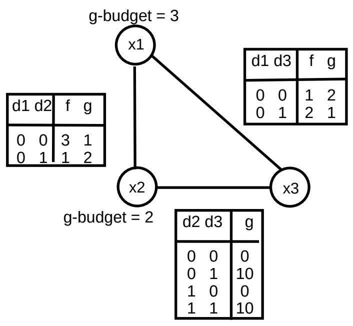


Figure 7: MCASA fails on non-T-nodes

As we can see in Figure 7, x_1 is a non-T-node since x_3 is a child of x_2 not x_1. If we assume that x_1 has a g-budget of 3 and only one value in its domain, then it must choose how to split its g between its two children. Based upon the functions on the links, it will choose to give a g-threshold of 2 to x_2 and 1 to x_3, leading to a predicted f-cost of 2 + 1 = 3. This effectively removes the value 0 from x_3's domain and causes the link between x_2 and x_3 to incur a g-cost of 10 which in turn leads x_2's g-constraint to be unsatisfiable for any g-threshold it could receive from x_1 (Since x_2 tries out both values under given threshold of 2). Since no matter what g-threshold it assigns to x_2, x_2's g-constraint is violated (leading to an infinite f-cost), x_1 infers that the problem is unsatisfiable. It is at this point that condition (3) from the previous proof has been violated since x_1 estimates $f_{current}(gt, x_2) = \infty$ whereas $f_{actual}(gt, x_2) = 4$ where $gt \in \{1, 2, 3\}$. This counter example is based on having just a g-function on the $x_2 - x_3$ link, however, similar examples can be constructed using just an f-function or both an f- and a g-function on the link. Modifications to MCASA to make these cases feasible is an issue for future work. ■

We now turn to MCA complexity. The original DCOP problem is NP-hard. By illustrating a solution to the multiply-constrained problem which only adds a linear number of additional nodes we suggest that the complexity class has not worsened. With respect to space, a key feature of Adopt is that its space complexity is linear in the number of nodes, n, specifically $|D_i|\, n$. In MCAP and MCAS, the space used at each regular node is the same, but we add up to n virtual variables, so the space complexity for MCAP and MCAS is $(|D_i| + 1)n$. In MCASA there are no virtual variables, but each node stores a g-to-f mapping for each of its children. This causes the space complexity to be $|D_i|\, n + G_i n$.

5. EXPERIMENTAL RESULTS

This section presents five sets of experiments. The first compares the performance of MCAP, MCAS and MCASA on four settings that were motivated by the domains in Section 2.3. Setting 1 com-

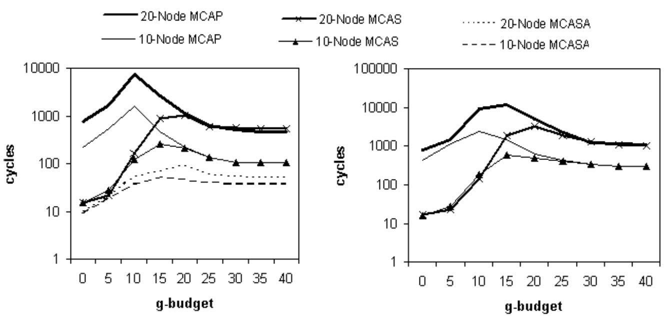


Figure 8: g-budget vs. run-time for a) 100% T-node problems b) 85% T-node problems

prised 20-node problems, with 3 values per node, an average link density of 1.9 and maximum link density of 4. It had 100% T-nodes and both the f- and g-costs were randomly chosen from a uniform distribution varying from 0 to 10. Setting 2 is similar to setting 1, except that the graph was 85% T-node (which increased the average link density to 2.2) to allow for comparison of the impact of T-nodes. Setting 3 (setting 4) is similar to setting 1 (setting 2), except that it is a 10-node problem. We created 15 sets of constraint functions for each of our domain settings, i.e. each data-point in our graphs is an average of 15 problem instances.

To highlight the tradeoff between the techniques, we show the performance of each technique when applied to all the nodes in a problem i.e. we either apply MCAP to all the nodes or MCAS or MCASA. There are a total of 1350 different runs summarized in the two graphs given all the g-budgets tested. Figure 8a shows the average run-times of MCAP, MCAS, MCASA in settings 1 and 3. The x-axis shows the g-budget applied to all variables and ranges from 0, which is unsatisfiable, to 40, which is effectively a singly-constrained problem. The y-axis shows runtime – run-time is measured in cycles where one cycle consists of all agents receiving incoming messages, performing local processing and sending outgoing messages [9]. The y-axis is logarithmically scaled.

The graphs show that due to privacy, MCAP has the poorest performance. The upper bounds calculated by sharing information in MCAS improve performance, while the exact bounds and lack of tree-restructuring in MCASA give it the best performance. The maximum speedup of MCAS over MCAP is 744 for setting 1 and 209 for setting 3, while the maximum speedup of MCASA over MCAS is 937 for setting 1, and 190 for setting 3. Figure 8b demonstrates similar results for setting 2 and setting 4. Only MCAP and MCAS results are shown given the switch to 85% T-node problems in these settings (we cannot apply MCASA to all the nodes in these settings). The switch from 100% T-node to 85% T-nodes causes a significant increase in run-time (note the y-axes are not identical).

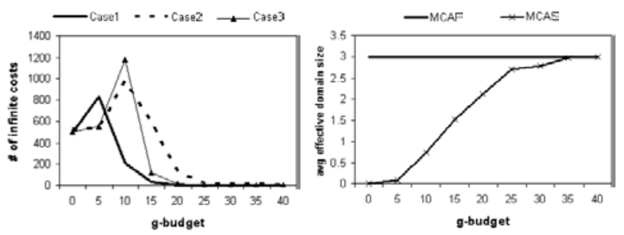


Figure 9: a) g-budget vs. number of infinite cost messages b) g-budget vs. number of values per domain

For all of the subalgorithms the runtime curve has a distinct inverted U-shape to it: the run-times are lowest at high g-budgets (no resource constraints) or at low g-budgets (tight resource constraints). Figure 9a begins to provide an explanation by plotting the g-budget on the x-axis and the total number of infinite cost messages received by any variable in the problem – due to g-constraint violation – on the y-axis. The figure shows results from three representative runs of MCAP on setting 3. We can see the same hump appearing at a g-budget of 10 and diminishing to almost 0 at a g-budget of 20. This agrees with the fact that the maximum g-cost on a link is 10 and the average link density is 1.9 (there are still one or two infinite cost messages all the way up to a g-budget of 35 because the maximum link density is 4). The larger number of infinite cost messages in the mid-g range indicates that it potentially takes longer to discover unsatisfiability of solutions, leading to longer run-times and hence the hump shape.

In all settings, for tighter g-budgets, the MCAS algorithm outperforms MCAP, however, for looser g-budgets, there is no difference in performance. Figure 9b provides an explanation based on the fact that MCAS sends g-thresholds that potentially prune descendents' domain values. We plot the g-budget on the x-axis and the average number of values remaining in the domain of a variable running MCAS on the y-axis. The numbers are plotted as an average over all nodes over all 15 problem instances of setting 3. For comparison, we also provide results of MCAP, which performs no pruning and hence is a flat line. This figure shows that when g-budgets are tight, MCAS provides significant pruning, but when g-budgets are loose MCAS upper-bounds provide no pruning (in comparison with MCAP) and thus there is no efficiency loss due to privacy. The domain sizes converge at a g-budget of 25, which is also where the runtimes converge in Figure 8.

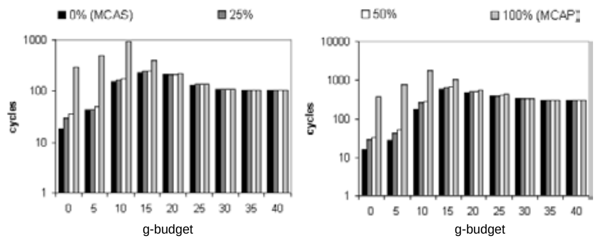


Figure 10: g-budget vs. runtime, varying percentages of private constraints: a) 100% T-node and b) 85% T-node problems

While we have performed several other explorations of privacy-efficiency tradeoffs in our techniques, space limitations preclude an extensive discussion. For example, one set of experiments demonstrate the benefits of the per-node application of the MCAP and MCAS techniques. Here, we took the examples from settings 1 and 2 from Figure 8a and b and randomly assigned first 25% and then 50% of the nodes to have private g-constraints while the remaining were assumed to be non-private. We then compared their performance to that of MCAP (100% private) and MCAS (0% private). In order to isolate the effect of privacy, not graph structure, we did not exploit T-nodes. The results are shown in Figure 10a and b. The x-axis again shows the g-budget applied and the y-axis measures the runtime in cycles on a logarithmic scale. Each bar in the graph shows an average over the 15 instances and we can see that as the percentage of nodes whose additional constraint is private increases, the runtime increases for smaller g-budgets. Interestingly, the increase in run-time is not proportional to the increase in the number of private g-constraints; there is a significant jump in run-time when all nodes have private g-constraints. However, as in Figure 8, when the g-budget on each variable is loose enough the runtimes converge because no pruning takes place.

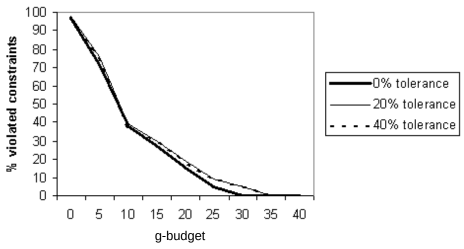


Figure 11: g-constraint violation by Adopt

While we developed MCA, one question was whether singly-constrained algorithms could automatically satisfy the g-constraint simply by loosening their tolerances on optimal solutions. We examined this by measuring the degree to which the g-constraint is violated by Adopt when ignoring g but specifying a tolerance on f. Figure 11 shows the results with each data point representing an average over 5 randomly generated runs. The x-axis gives the g-budget applied to each variable and again varies from 0 to 40. The y-axis is the percentage of nodes whose g-constraint was not satisfied by the solution obtained by Adopt. The tolerance is a percentage of the difference between the optimal and maximum cost solutions. The key result is that even with a large 40% tolerance on f, large numbers of g-constraints are violated with small g-budgets. Thus, running singly-constrained algorithms with tolerances is inadequate in addressing multiply-constrained DCOPs.

6. CONCLUSION AND RELATED WORK

This paper introduces a novel algorithm for multiply-constrained DCOPs, where agents must not only optimize a global objective, but must also satisfy resource constraints within their local neighborhoods. These resource constraints may be defined over an overlapping set of the variables in the original optimization constraints and they may be private constraints. Leading DCOP algorithms of today [9, 11, 14] are unable to fully address multiply-constrained DCOP problems; in particular, they are unable to exploit the interaction between the searches for the global optimal and local satisfaction, or address the privacy-efficiency tradeoffs engendered. To remedy this situation, this paper presents a novel multiply-constrained DCOP algorithm based on mutually-intervening search, which is operationalized via three ideas: (i) transforming the constraint graph to allow private n-ary constraints to be enforced; (ii) revealing upper-bounds on (resource) costs to neighbors, in order to gain efficiency while sacrificing privacy; (iii) identifying a local graph structure property – T-nodes – which allows agents to gain further efficiency by providing exact bounds on resource costs to neighbors. These ideas were realized via the MCA algorithm, which builds on Adopt, currently one of the most efficient DCOP algorithms. We proved the correctness of MCA, and presented detailed experimental results, illustrating the profile of the privacy-efficiency tradeoffs.

In terms of related work, we have already discussed the relationship of multiply-constrained DCOP and MCA with singly-constrained DCOP algorithms, such as Adopt, OptAPO and DPOP [9, 11, 14]. While we build on top of Adopt, we recognize that no DCOP algorithm currently dominates all others across all domains. However, the techniques developed here would transfer to other algorithms, e.g. MCAS and MCASA style techniques could be applied to algorithms like OptAPO. Whether DPOP [11] could similarly benefit from our techniques is an issue for future work.

Also related is previous work in multi-criteria collaboration [7, 10] which looks at finding a pareto-optimal for multiple objectives rather than optimizing a single objective subject to resource constraints. Furthermore, unlike our research, that work does not build on the recent theoretical or algorithmic advances in DCOPs. Approaches to multi-criteria constraint satisfaction and optimization problems have tackled the problem using centralized methods [4], but we focus on a distributed problem, which requires designing algorithms where agents function without global knowledge.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010. We would like to thank Barbara Grosz, Hyuckchul Jung and Jay Modi for their valuable feedback.

8. REFERENCES

[1] S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS*, 2005.

[2] J. Cox and E. Durfee. A distributed framework for solving the multiagent plan coordination problem. In *AAMAS*, 2005.

[3] J. A. Espinosa and E. Carmel. The impact of time separation on coordination in global software teams: a conceptual foundation. *SOFTWARE PROCESS IMPROVEMENT AND PRACTICE*, 8, 2004.

[4] M. Gavanelli. An algorithm for multi-criteria optimization in CSPs. In *ECAI*, pages 136–140, 2002.

[5] P. Jalote and G. Jain. Assigning tasks in a 24-hour software development model. In *Asia-Pacific Software Engineering Conference (APSEC04)*, 2004.

[6] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, 2004.

[7] N. Matsatsinis and P. Delias. A multi-criteria protocol for multi-agent negotiations. *Lecture Notes in Computer Science*, 3025, 2004.

[8] A. Meisels and O. Lavee. Using additional information in discsps search. In *Distributed Constraint Reasoning Workshop (DCR)*, 2004.

[9] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence Journal*, 161:149–180, 2005.

[10] P. Moraitis and A. Tsoukias. A multicriteria approach for distributed planning and negotiation in multiagent systems. In *ICMAS*, 1996.

[11] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, Edinburgh, Scotland, Aug 2005.

[12] M. Silaghi and D. Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. In *(IAT)*, 2004.

[13] K. Stergiou and T. Walsh. Encoding non-binary constraint satisfaction problems. In *AAAI*, 1999.

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

[15] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP*, 2002.