Multiply-Constrained DCOP for Distributed Planning and Scheduling

Emma Bowring and **Milind Tambe**
Computer Science Dept.
University of Southern California
Los Angeles CA 90089
{bowring,tambe}@usc.edu

Makoto 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 planning and scheduling. 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 local 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. This algorithm is based on mutually-intervening search, i.e. using local resource constraints to intervene in the search for the optimal solution and vice versa, realized via three key ideas: (i) transforming n-ary constraints via virtual variables 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.

Introduction

Distributed Constraint Optimization (DCOP)(Modi *et al.* 2005; Mailler & Lesser 2004; Petcu & Faltings 2005) is a useful technique for applications involving multiagent planning and scheduling, e.g. distributed meeting scheduling, distributed factory and staff scheduling, and sensor net scheduling(Meisels & Lavee 2004; Hanne & Nickel 2003). In a DCOP, 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 (Modi *et al.* 2005; Mailler & Lesser 2004; Petcu & Faltings 2005), 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, fuel, transportation. These resource constraints necessitate DCOP algorithms that optimize a global objective, while ensuring that resource limits aren't exceeded. While in some domains agent must keep these resource constraints private (e.g. travel budgets in distributed meeting scheduling(Meisels & Lavee 2004)), in others resource constraints may be non-private (e.g. overtime limits in staff allocation for distributed software development).

Thus, there is a need for multiply-constrained DCOP algorithms. There are three primary challenges in designing such algorithms. First, agents' additional resource constraints add to DCOP search complexity. Hence, agents must quickly prune unproductive search paths. Second, harnessing state-of-the-art DCOP algorithms is crucial (henceforth, we refer to these as "singly-constrained" DCOP algorithms) given their efficiency and continual algorithmic improvements. This is challenging because the additional resource constraints are local, possibly private and defined over different domains. Third, algorithms must exploit constraint revelation to gain efficiency when privacy is not crucial. (Unlike (Yokoo, Suzuki, & Hirayama 2002) which uses cryptographic techniques in DisCSP/DCOP for privacy, we do not insist on such watertight privacy, as the use of multiple external servers may not always be desirable.)

This paper presents a novel multiply-constrained DCOP algorithm that 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 and opportunistic search for the global optimal solution obviates testing all partial solutions for resource constraint satisfaction. There are three key techniques in this algorithm which are facets of mutually-intervening search: constraint transformation, dynamically-constraining search and local acyclicity. The first idea, *constraint transformation*, adds a virtual variable v' to the original DCOP problem to represent the resource constraint at each variable v. Variable v' is owned by the same agent as v and provides high negative utility if its neighbors' values violate the resource-constraint tested by v'. These virtual variables enable singly-constrained algorithms to be exploited for multiply-constrained DCOPs and, by manipulating virtual variable placement in the DCOP graph, constraint privacy is maintained. The next two ideas focus on 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 pre-select val-

ues 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 (Modi *et al.* 2005), our techniques could be applied to other algorithms, e.g. OptAPO and SynchBB (Mailler & Lesser 2004; Yokoo *et al.* 1998). 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. The different techniques can be applied simultaneously to different nodes in the same problem. We experimentally compare these techniques and illustrate problems where agents may gain the most efficiency by sharing resource constraints and problems where agents lose no efficiency by maintaining privacy.

Problem Definition

DCOP

A DCOP(Modi *et al.* 2005; Petcu & Faltings 2005) 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 a set of constraints and associated cost functions, $f_{ij} : D_i \times D_j \rightarrow N \cup \infty$, 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 $x_i \leftarrow d_i, x_j \leftarrow d_j \in A$. Here, variables x_i and x_j are considered neighbors since they share a constraint.

Taking as an example the constraint graph in Figure 1 where x_1, x_2, x_3, and x_4 are variables each with domain {0,1} and the f-cost functions shown (ignoring g), $F((x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0)) = 4$ and the optimal would be $(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)$. While the above commonly used definition of DCOP emphasizes binary constraints, *general DCOP representations* may include n-ary constraints.

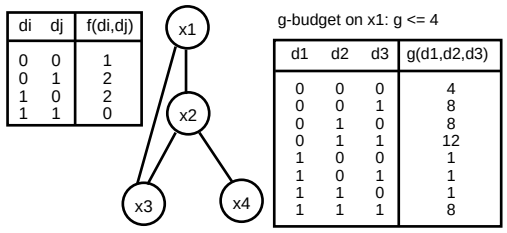

Figure 1: Multiply-Constrained constraint graph

Multiply-Constrained DCOP

In Multiply-Constrained DCOP we make use of n-ary DCOP constraints by adding an new cost function g_i on a subset of x_i's links and a g-budget G_i that 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, 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 where x_i's neighbors. We define g-constraints to be private or shared. If the 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 n-ary (g-cost) constraints, f- and g-constraints defined over overlapping sets of variables, privacy of g-constraints, and the need to exploit the interaction between f- and g-constraints for pruning the search space. No current DCOP algorithm, including leading algorithms such as Adopt, OptAPO and DPOP(Modi *et al.* 2005; Mailler & Lesser 2004; Petcu & Faltings 2005), are able to address all these issues together.

Motivating Domains

In this section we demonstrate the need for multiply-constrained DCOP algorithms in distributed planning and scheduling with two examples.

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 a simple example shown in Figure 2, where researchers, A, B, C, D and E are divided between organizations at Loc1 and Loc2. A and C need to meet, and B, D and E (E's attendance is not mandatory) also need to meet. A and C are group leaders who manage travel budgets for their groups and wish to keep budgetary constraints private.

Multiply-constrained DCOP allows us to model this problem. We assume that agents share all their scheduling preferences with their appropriate neighbors but not their budgetary constraints. Thus, our representation is a generalization of MAP(Modi & Veloso 2005). The domain of each variable is a tuple of time-of-meeting and location, e.g. {6th

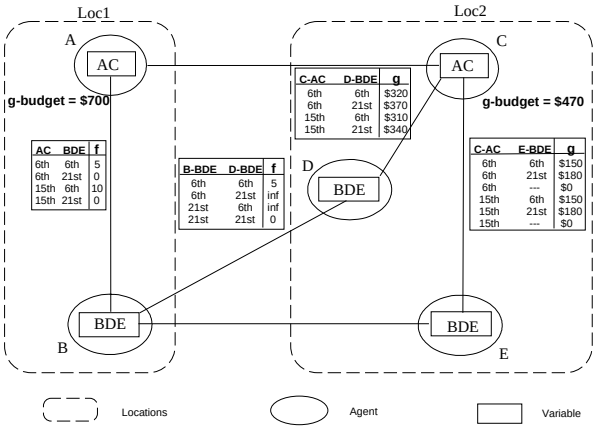

Figure 2: Meeting Scheduling Example

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 BDE 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 associated with subgroups of agents reflects their travel cost. 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, D or E in either the AC or BDE meeting if they are held at Loc1. A and C may not wish to reveal their travel 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 (Espinosa & Carmel 2004; Jalote & Jain 2004). While interdependent tasks within a project must be scheduled for their timely completion, a team liaison must videoconference with the team to which the code is being sent to facilitate the handoff. However, this liaison may have to stay after hours to teleconference with the new team in its time zone. Figure 2 shows an example involving five tasks $\{t_1, \ldots, t_5\}$, where 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.

The two examples above illustrate the need for multiply-constrained DCOPs to model such collaborations. These domains require agents to optimize an f-cost function and yet adhere to (private) g-constraints.

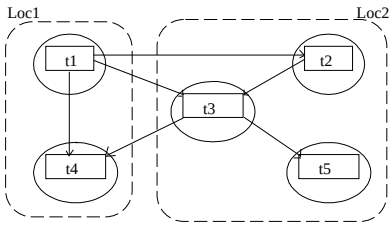

Figure 3: Software Development Example

Background: ADOPT

Adopt(Modi *et al.* 2005) is an asynchronous complete DCOP algorithm. Adopt has been discussed in detail in several publications(Modi *et al.* 2005; Cox & Durfee 2005), so we provide only a brief description here. Adopt organizes agents 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. While for expository purposes it is useful to consider each variable as belonging to a separate agent, Adopt does not require a single variable per agent.

Adopt employs two basic messages: VALUE and COST.[1] Assignments of values to variables are conveyed in VALUE messages that are sent to variables sharing a constraint with the sender, lower in the DFS tree. 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.

Multiply-Constrained Adopt (MCA)

Basic Ideas

In this section we discuss the basic ideas that go into forming the mutually-intervening searches of the MCA algorithm. Mutually-intervening search addresses a major challenge of multiply-constrained DCOP (namely greater complexity) by

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

```
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) ← Neighbors(x_i) ∪ x_i
(5)       Neighbors(x_i) ← Neighbors(x_i) ∪ x'_i
(6)       forall x_k ∈ Neighbors(x_i)
(7)         if x_k is not a neighbor of ≥ one x_l ∈ Neighbors(x_i)
(8)           Neighbors(x_k) ← Neighbors(x_k) ∪ x_l
(9)       rebuildDFStree(x_1 ... x_n)
(10)  forall x'_i parent(x'_i) ← lowest priority Neighbor of x'_i

Initialize
(11)  CurrentContext ← {}
(12)  initialize structures to store lb and ub from children
(13)  d_i ← d that minimizes LB(d)
(14)  if private_i == false and Tnode_i == true
(15)    forall x_l ∈ Children
(16)      gThresh_i(x_l) ← 0
(17)      for gt ← 0 ... G_i
(18)        GFmap(x_l, gt) ← min f(d_l, d_i) s.t. g(d_l, d_i) ≤ gt
(19)  backTrack

when received (VALUE, x_j, d_j, gThresh_j)
(20)  if private_j == false
(21)    add (x_j,d_j,gThresh_j) to CurrentContext
(22)  else add (x_j,d_j) to CurrentContext
(23)  reset lb and ub if context has changed
(24)  if private_i == false
(25)    forall x_l ∈ Children
(26)      if gContext(x_l) incompatible with CurrentContext:
(27)        gThresh(x_l) ← 0
(28)  backTrack;

when received (COST, x_k, context, lb, ub)
(29)  update CurrentContext
(30)  forall d' ∈ D_i, x_l ∈ Children
(31)    if context(d', x_l) incompatible with CurrentContext:
(32)      reset lb and ub
(33)  if context compatible with CurrentContext:
(34)    store lb and ub
(35)    GFmap(x_l, gThresh_il) ← lb s.t.
          (x_i, d_i, gThresh_il) is part of context from x_k
(36)  backTrack

procedure backTrack
(37)  if x_i not a virtual variable
(38)    if LB(d_i) > LB(d) for some d
(39)      d_i ← d that minimizes LB(d) and
            satisfies gThresh_j where x_j ∈ Ancestors(x_i) and
            private_j == false
(40)      if private_i == false
(41)        calcOptimalSplit
(42)        SEND (VALUE, (x_i, d_i, gThresh(x_k)))
            to each lower priority neighbor x_k
(43)      else SEND (VALUE, (x_i, d_i))
            to each lower priority neighbor x_k
(44)    if LB == UB:
(45)      if TERMINATE received from parent or x_i is root:
(46)        SEND TERMINATE to each child
(47)        Terminate execution;
(48)    if Tnode_parent == false
(49)      SEND (COST, x_i, CurrentContext, LB, UB)
(50)    else
(51)      SEND (COST, x_i, CurrentContext, LB, UB)
            to parent for each g we have tried g ≤ gThresh_parent
(52)  else
(53)    if g(d_i, CurrentContext) > gConstraint(x_i)
(54)      SEND (COST, x_i, CurrentContext, ∞, ∞) to parent
(55)    else SEND (COST, x_i, CurrentContext, 0, 0) to parent
```

Figure 4: Multiply-Constrained Adopt Pseudo-code

having agents immediately intervene in the search for the optimal f-cost if its g-constraint is violated and conversely allowing quick pruning of the high f-cost solutions to obviate checking if such an assignment violates a node's g-constraint. This overall approach is realized via three basic techniques: constraint transformation, dynamically constraining search and local acyclicity. We now address each of these in more detail.

Constraint Transformation To harness the efficiency of existing DCOP algorithms and maintain privacy, we add virtual variables to enforce g-constraints, e.g. for the problem described in Figure 1, we add x'_1 to represent an n-ary constraint between x_1, x_2, x_3. Virtual variables are controlled by the original variable's owner and enforce the original variable's g-constraint by sending infinite costs when a constraint is violated to the nodes involved, preemptively cutting off that line of search. Since typical DCOP algorithms such as Adopt are based on binary cost functions that are shared (and not private) across agents, n-ary constraints require additional mechanisms within such algorithms. By encapsulating the n-ary constraint in a virtual variable and placing the virtual variables as leaves in the DFS tree, we can protect the privacy of both the g-function and the g-budget since it is encapsulated inside the virtual variable and not present on any of the links.

Dynamically Constraining Search When privacy is not essential, it is important to exploit g-constraint revelation to gain efficiency in the f-optimization and vice versa. We address the first aspect by requiring descendant nodes to only consider assignments that will not violate the g-constraints of their ancestors. Specifically we pass each descendants a bound (termed g-threshold) specifying how large a g-cost they can pass up, thus limiting their choice of domain values. This g-threshold can represent an exact bound when the local graph is acyclic or an upper bound otherwise. If a value combination fails to satisfy an ancestor's g-constraint, descendants will not explore to see whether this value combination is optimal in f. Additionally, the opportunistic search for an optimal f means that variables will only try to enforce their g-constraint for those value combinations that seem to be of low f-cost. Thus, the searches dynamically constrain each other, leading to performance improvements.

Local Acyclicity (T-nodes) The notion of locally acyclic graph structure is captured more formally via our definition of T-nodes given below. Variable x_i is a T-node if all lower priority g-constrained neighbors of x_i are in separate branches of the DFS tree. In our original DCOP example in Figure 1, x_1 is not a T-node because two of its descendents are in the same subtree (x_2 and x_3), but x_2, x_3 and x_4 are all T-nodes. Identifying local acyclicity is important because it allows us to treat our children as independent and this allows for calculating exact (rather than upper) bounds for the dynamically constraining search.

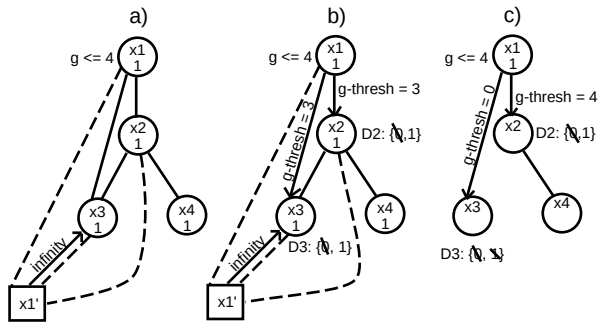

Figure 5: a) MCAP b) MCAS c) MCASA

Algorithm Description

Figure 4 presents pseudo-code for the MCA algorithm. [2] The following conventions are used: (i) x_i denotes the variable itself, (ii) x_j denotes a higher priority neighbor, and (iii) x_k and x_l denote lower priority neighbors. To emphasize the new elements we have included only general descriptions of functionality that remains unchanged from the original Adopt algorithm(Modi *et al.* 2005). There are three separate techniques used in MCA, and we assign each technique a name: MCA Private (MCAP) is employed when a constraint may not be revealed to its neighbors and it employs constraint transformation, MCA Shared (MCAS) is used when a constraint is sharable but the variable is not a T-node and it employs constraint transformation as well as dynamically constraining search, and MCA Shared and Acyclic (MCASA) is utilized when a constraint is sharable and the variable is a T-node and it relies solely on dynamically constraining search. We take privacy to mean that neither the g-functions nor the g-budgets are explicitly revealed to any other agent.

We will first discuss MCAP. An example snapshot of its execution on the problem from Figure 1 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 very high cost (infinite cost) is sent to the cluster of nodes involved. 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 original and the virtual variable are controlled by the same agent. The feedback is sent to the lowest priority of the variables involved in the broken constraint (x_3). Using the f-cost optimization mechanisms of Adopt, the feedback will be propagated to the other node(s) that must change their values to find the optimal solution that still satisfies the g-constraints.

Since feedback must be able to be sent to all the nodes in a constraint, Adopt's DFS tree must be restructured to put all the variables involved in the same subtree (lines 6-9 in figure 4). MCA preprocessing creates the virtual variables which need to be placed as leaves lower in the DFS tree than the variables in the g-constraint they enforce (line 10). A virtual variable will only receive VALUE messages and will then determine whether the current assignment violates the g-constraint it represents. If so, an infinite cost is passed up, forcing the variables to try a different assignment, otherwise a cost of 0 is passed up to the appropriate variable (lines 53-55).

While feedback signals have the advantage of maintaining the privacy of the g-constraints and the variables involved in them by encapsulating the g-cost functions and g-budget inside a variable, it has a drawback: its partial search of unsatisfying assignments slows the algorithm. When a g-constraint – the specific g-functions and the g-budget – is non-private, we can reveal the g-function of a link to those vertices connected to it, which is the same privacy loss tolerated for the f-function. With this information we can implement MCAS and MCASA. Snapshots of MCAS and MCASA are shown in Figures 5b and 5c respectively.

In MCAS and MCASA we exploit the sharing of the g-functions by having parents send their descendants g-thresholds in the VALUE messages (lines 20-22) indicating that the child 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 node x_1 to nodes x_2 and x_3. (Note that as discussed earlier, we make use of the assumption that the g-functions are additive here.) If the variable is not a T-node (in MCAS), the g-thresholds represent an upper bound, constituting the total g-budget minus the minimum g-cost that could be consumed by each of the other links involved:
$(G_i - \sum_{x_j \in Neighbors(x_i) \neq x_l} min\ g_{ij}(d_i, d_j))$. In the example in Figure 5b, the total g-budget on x_1 is 4, and the minimum usable on each link is 1, and hence the messages set a g-threshold upper-bound of 3 for each child. Given this upper-bound, a node can prune out certain values from its domain as violating the g-constraint (e.g. 0 is pruned from the domain of x_3) leading to speedups over MCAP. For T-nodes it is possible to calculate an exact bound (MCASA)—in Figure 5c, the exact bound is 0 for node x_3 and 4 for node x_2 – enabling more values to be pruned out, further speeding up the search. Note that Figure 5c is slightly modified from the original example to show T-nodes. Calculating the exact bounds requires a node to maintain a mapping from potential g-thresholds to f-costs (GFmap) for each of its lower priority neighbors. This is initially constructed from the link function (line 17) and then updated as COST messages arrive (line 35). A T-node uses these g-threshold to f-cost lb mappings for each of its descendents to calculate how to split its g-budget among them. This calcOptimalSplit function can be implemented using a straightforward dynamic program and thus is omitted from the pseudo-code. The new g-thresholds engender a design choice. Nodes can either store lb and ub information about f-costs for each possible g-threshold as they do for each of their own values or they can consider the g-thresholds part of their context and restart their local optimization whenever the g-thresholds change. The former option is simpler and possibly more efficient in terms of cycles but it increases the space requirements of each node so we have chosen to implement the latter approach.

[2] A more up-to-date version of the pseudocode is available in the proceedings of AAMAS 2006

Correctness of Multiply-Constrained Adopt

In this section we will again separate out the proofs for each technique for the sake of clarity. 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:*

$OPT(x_i, context) \stackrel{def}{=}$
$min_{d \in D_i}[\delta(d) + \sum_{x_l} OPT(x_l, context \cup (x_i, d))]$
if $g_i(d_i, \{d_j | x_j \in Neighbors(x_i)\}) \leq G_i$
∞ *otherwise*
where $\delta(d) \stackrel{def}{=} \sum_{x_j \in ancestors} f_{ij}(d_i, d_j)$

Proof: To show this we start from the correctness of the original Adopt algorithm which was proven in (Modi *et al.* 2005). Original Adopt will find at every node x_i:

$OPT'(x_i, context) \stackrel{def}{=}$
$min_{d \in D_i}[\delta'(d) + \sum_{x_l} OPT'(x_l, context \cup (x_i, d))]$
where $\delta'(d) \stackrel{def}{=} \sum_{x_j \in ancestors} f_{ij}(d_i, d_j)$

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 nodes 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: This requirement easily follows from the fact that 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 and discuss the exact bounds communicated to children of T-nodes in MCASA. 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 finds 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.*

Proof by Contradiction: Assume that there exists a g split: $g_k\ \forall x_k \in Children(x_i)$ which is not optimal, but which seems to be based on current incomplete information. Thus there exists another split: $g_k^* \forall x_k \in Children(x_i)$ which is optimal but which currently seems not to be. If f_{actual} is the cost when full knowledge is attained and $f_{current}$ is the current lower bound on cost and $f(g')$ is the f-cost accumulated when employing split g':

1. $\sum_{x_k} f_{actual}(g_k) > \sum_{x_k} f_{actual}(g_k^*)$
2. $\sum_{x_k} f_{current}(g_k) < \sum_{x_k} f_{current}(g_k^*)$
3. $\sum_{x_k} f_{current}(g'_k) \leq \sum_{x_k} f_{actual}(g'_k)$ for any split g'_k

Thus x_i will currently choose to employ split $g_k \forall x_k \in Children(x_i)$. Since there are a finite number of nodes in the tree below x_i, at some point before termination $\sum_{x_k \in Children(x_i)} f_{current}(g_k) = \sum_{x_k \in Children(x_i)} f_{actual}(g_k)$ will become true.

Now assume that MCASA terminates without switching from g_k to g_k*. If MCASA continues to use split g_k then at this point:

$$\sum_{x_k} f_{current}(g_k) = \sum_{x_k} f_{actual}(g_k)$$
$$\sum_{x_k} f_{current}(g_k) > \sum_{x_k} f_{actual}(g_k^*) \quad \text{by (1)}$$
$$\sum_{x_k} f_{current}(g_k) > \sum_{x_k} f_{current}(g_k^*) \quad \text{by (3)}$$

Thus based upon current information ($f_{current}$) MCASA will switch from g_k to g_k* because it has a lower associated f-cost. This contradicts our assumption that MCASA will not switch splits before terminating. ■

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 by counter-example.

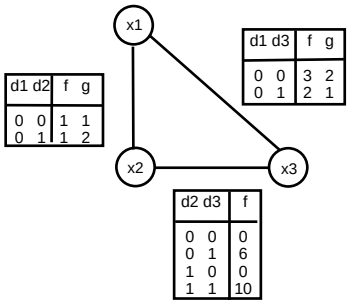

Figure 6: MCASA fails on non-T-nodes

As we can see in Figure 6 x_1 is a non-T-node since its two children x_2 and x_3 have a constraint between them. 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 1 + 2 = 3. This effectively removes the value 0 from x_3's domain. x_2 now tries the value of 1, to obtain an f-cost of 13, and then tries the value of 0 (which is within its g-threshold) to obtain an f-cost of 10. Hence, x_2 sends x_1 cost messages indicating that the entries in its table should be increased to f(g=1) = 10 and f(g=2) = 10. However, since x_2 will return an f of 10 regardless of g of 1 or 2, and x_3 will return an f of 2 for any g, x_1 see no reason to change its current allocation of g-thresholds.

■

Experimental Results

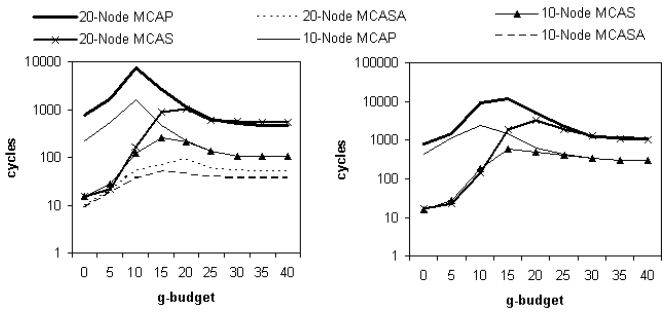

Figure 7: g-budget vs. run-time for a) 100% T-node problems b) 85% T-node problems

The first experiment compares the performance of our different techniques, MCAP, MCAS and MCASA on four separate domain settings that have been motivated by the domains presented in Section 2.3. Setting 1 focused on 20-node problems, each with 3 values per node, with an average link density of 1.9 and maximum link density of 4. This graph had 100% T-nodes to emphasize the speedups due to MCASA. In this setting, both the f- and g-costs were randomly chosen from a uniform distribution varying from 0 to 10. The g-constraint was assumed to be an additive function (see Section 3). Setting 2 is similar to setting 1, except that the graph was 85% T-node (which caused the average link density to increase to 2.2) to emphasize the speedups due to MCASA. Setting 3 (setting 4) is similar to setting 1, except 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 really highlight the tradeoff between the different 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, and thus, there are a total of 1350 different runs summarized in this first experiment. Figure 7a shows the average run-times of the MCA sub-algorithms over 15 instances in settings 1 and 3, for different g-budgets. 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 on log-scale — as with other DCOP systems, run-time is measured in cycles of execution where one cycle consists of all agents receiving incoming messages, performing local processing and sending outgoing messages(Modi *et al.* 2005).

The graphs show that the run-times of all the techniques are lowest when the search is either unconstrained due to a high g-budget or extremely constrained due to a low g-budget. Due to the privacy requirement, 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 MCAP is 750 for setting 1, and 216 for setting 3. Figure 7b 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 also causes a significant increase in runtime (note the y-axes are not identical). In all settings, for tighter g-budgets, the MCAS algorithm outperforms MCAP, however, for looser g-budgets, there is no difference in performance: when g-budgets are loose MCAS upper-bounds provide no pruning and thus there is no efficiency loss due to privacy.

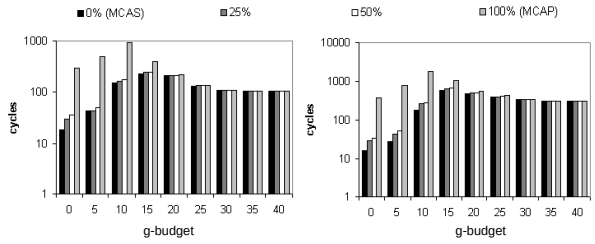

Figure 8: g-budget vs. runtime for differing percentages of private constraints on a) 100% T-node problems and b) 85% T-node problems

In order to demonstrate the benefits of the per-node application of the MCAP and MCAS techniques, we took the examples settings 1 and 2 from Figure 7a 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). The results are shown in Figure 8a 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. How-

ever, as was found when comparing MCAS against MCAP in the previous figure, when the g-budget on each variable is loose enough the runtimes converge because no pruning takes place as a result.

Conclusion and Related Work

While DCOP is rapidly emerging as a key technology for multiagent planning and scheduling(Modi *et al.* 2005; Mailler & Lesser 2004; Petcu & Faltings 2005), previous DCOP work suffers from the limitation of optimizing a single global objective. In many real-world distributed planning and scheduling domains, however, agents must locally (within their local neighborhood) satisfy additional resource constraints. Demands from real-world domains for resource constraints has led to our new formulation of multiply-constrained DCOPs, where agents are provided additional (resource) constraints that they must satisfy. This paper provides a novel multiply-constrained DCOP algorithm, based on the overarching theme of mutually-intervening searches, which is operationalized via three key ideas: (i) transforming n-ary constraints via virtual variables to maintain privacy while harnessing existing singly-constrained DCOP algorithms; (ii) revealing upper-bounds on (resource) costs to neighbors, in order to gain efficiency while sacrificing privacy where permitted; (iii) identifying a local graph structure property – T-nodes – which allows agents to gain further efficiency by providing not just upper-bounds but exact bounds on resource costs to neighbors. These ideas were realized in the context of Adopt, one of the most efficient current DCOP algorithms. The Multiply-Constrained Adopt (MCA) algorithm modulates its performance based on the privacy requirements of individual constraints and the local network structure. We proved the correctness of the MCA algorithm, and presented detailed experimental results, illustrating the profile of the privacy-efficiency tradeoffs in MCA, and the benefits of T-nodes.

In terms of related work, there is significant continued progress in singly-constrained DCOP algorithms (Modi *et al.* 2005; Yokoo *et al.* 1998; Petcu & Faltings 2005), e.g. OptAPO and DPOP have been shown to compare favorably with Adopt in some domains and vice versa(Mailler & Lesser 2004; Davin & Modi 2005; Petcu & Faltings 2005) and ultimately, the costs of computation vs. communication in the domain may govern the choice of an appropriate algorithm. Our work is complementary to these advances. First, the multiply-constrained DCOP formulation presents a new challenge for all of these algorithms. Furthermore, some of the techniques developed here would transfer to these other algorithms, e.g. MCAS and MCASA style techniques could be applied to algorithms like OptAPO. Whether the DPOP algorithm(Petcu & Faltings 2005) will similarly benefit from the techniques introduced here is a challenge for future work – indeed, variable elimination techniques such as DPOP may face significant challenges when addressing multiply-constrained DCOPs. Approaches to multi-criteria constraint satisfaction and optimization problems have tackled the problem using centralized methods (Gavanelli 2002), but our central contribution is in tackling a distributed problem, which requires designing algorithms where agents function without global knowledge.

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.

References

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

Davin, J., and Modi, P. 2005. Impact of problem centralization in distributed constraint optimization algorithms. In *AAMAS*.

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

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

Hanne, T., and Nickel, S. 2003. A multi-objective evolutionary algorithm for scheduling and inspection planning in software development projects. *Institute for Technical and Economic Mathematics (ITWM) Technical Report* 42.

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

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

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

Modi, P. J., and Veloso, M. 2005. Bumping strategies for the multiagent agreement problem. In *AAMAS*.

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

Petcu, A., and Faltings, B. 2005. A scalable method for multiagent constraint optimization. In *IJCAI*.

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

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