Balancing Local Resources and Global Goals in Multiply-Constrained DCOP*

Emma Bowring
Computer Science Department
University of the Pacific
Stockton, CA 95211
USA

Milind Tambe
Computer Science Department
University of Southern California
Los Angeles, CA 90089
USA

Makoto Yokoo
Computer Science Department
Kyushu University
Fukuoka, 819-0395
Japan

April 18, 2010

Abstract

Distributed constraint optimization (DCOP) is a useful framework for cooperative multiagent coordination. DCOP focuses on optimizing a single team objective. However, in many domains, agents must satisfy constraints on resources consumed locally while optimizing the team goal. Yet, these resource constraints may need to be kept private. Designing DCOP algorithms for these domains requires managing complex trade-offs in completeness, scalability, privacy and efficiency.

This article defines the multiply-constrained DCOP (MC-DCOP) framework and provides complete (globally optimal) and incomplete (locally optimal) algorithms for solving MC-DCOP problems. Complete algorithms

---

*This paper significantly extends our previous conference paper [3]. It extends that paper by providing all the incomplete algorithms from [2], as well as additional algorithms, more detailed explanation of these new algorithms, additional experimental results and analysis, and a detailed discussion of related work.
find the best allocation of scarce resources while optimizing the team objective, while incomplete algorithms are more scalable. The algorithms use four main techniques: (i) transforming constraints to maintain privacy; (ii) dynamically setting upper bounds on resource consumption; (iii) identifying the extent to which the local graph structure allows agents to compute exact bounds; and (iv) using a virtual assignment to flag problems rendered unsatisfiable by resource constraints. Proofs of correctness are presented for all algorithms. Experimental results illustrate the strengths and weaknesses of both the complete and incomplete algorithms.

1 Introduction

Various models have been proposed to handle cooperative multiagent coordination. These models differ in the complexity of the computation performed by individual agents and the complexity of the interactions among agents. Distributed Constraint Optimization (DCOP) [31, 27, 1, 38] is a cooperative multiagent coordination framework that has been applied to distributed meeting scheduling, distributed factory and staff scheduling, and sensor network domains [23, 30, 33, 19, 37].

While recent advances in efficient DCOP algorithms are encouraging [31, 27, 38, 1, 4, 46], there are complexities to real domains that are not captured by existing algorithms. This article focuses on the issues that arise from domains that require bounded optimization. Bounded optimization involves optimizing a global goal subject to a set of local restrictions; these restrictions may be externally imposed (resource-constrained DCOP) or internally imposed (semi-cooperative DCOP). An externally imposed bound is a feature of the problem (e.g. travel budgets in meeting scheduling problems) over which the agent has no control, whereas internally imposed constraints (e.g. time devoted to meetings) are within the control of the agent. An agent may have many reasons for imposing an internal constraint: individual interests, commitments to another team, or commitments to future goals of the team.

Bounded optimization domains require algorithms that can optimize a global objective, while ensuring that local resource limits are not exceeded. These algorithms must be able to keep the resource constraints private when required. Some bounded optimization problems require complete (globally-optimal) algorithms that can guarantee optimal usage of scarce resources. Other problems require extremely efficient incomplete (locally-optimal) algorithms that can handle large scale problems. Both types of algorithms are needed.

This article defines a new problem formulation called multiply-constrained DCOP (MC-DCOP), which captures bounded optimization problems. It then presents both complete and incomplete algorithms to solve MC-DCOP problems. There are four challenges to designing such algorithms:

• Search Complexity: Agents’ additional resource constraints add to the search complexity. The primary challenge is to handle the extra constraints without greatly increasing algorithmic runtime by rapidly pruning
search paths.

- **Harnessing Existing Algorithms:** We need to harness the efficiency of existing algorithms [31, 47, 50, 1, 38, 24] but, existing algorithms are not designed to handle constraints that can be private and not part of the global goal. We refer to these existing algorithms as singly-constrained DCOP algorithms to distinguish them from MC-DCOP algorithms.

- **Privacy/Efficiency:** Some domains require individual resource constraints to be kept private. The challenge is to handle private and non-private resource constraints simultaneously so that the inclusion of a single private constraint does not necessitate using a completely private algorithm.

- **Unsatisfiability Detection:** Since locally optimal algorithms only explore part of the search space, they need to know when to stop searching if resource constraints have rendered the problem unsatisfiable.

The complete Multiply-Constrained Adopt (MCA) algorithm presented in this article employs three techniques: constraint-graph transformation, dynamically-constraining search and local acyclicity. *Constraint-graph transformation* addresses the challenge of harnessing existing algorithms by employing virtual variables to enforce an agent’s resource constraints. Each virtual variable represents a single resource constraint and asynchronously notifies affected agents when an over-expenditure of resources occurs and thus preemptively prunes search paths. For correctness and privacy preservation, the algorithm restructures the multiagent network and appropriately places the virtual variables in that network. The other two techniques address the privacy/efficiency challenge. In *dynamically-constraining search*, an agent reveals to each neighbor an upper-bound on the resources available for that agent’s use. Ignoring potential solutions that are overly expensive improves algorithmic efficiency. *Local acyclicity* further improves efficiency in locally acyclic networks by allowing the resource bounds to be tightened to exact bounds. We show that these exact bounds cannot be applied in areas of the network that are not locally acyclic.

The incomplete Multiply-Constrained Maximum Gain Message algorithms (MC-MGM-1 and MC-MGM-2) presented in this article also make use of *constraint-graph transformation* and *dynamically-constraining search*. To address the unsatisfiability detection challenge, two design choices were available: (i) maintain a *satisfaction invariant* that resource limitations are never exceeded (similar to our avoidance of over-expenditure of resources in complete algorithms) and detect at the outset whether the problem is unsatisfiable or (ii) search for the optimal assignment without maintaining the invariant and detect during the search whether the problem is unsatisfiable. We used the former approach because it uses the resource limitations to prune the search space. In order to maintain the invariant, an initial satisfying assignment needs to be found, so a virtual starting assignment which met the invariant was added. The starting assignment was designed so that failure to move away from it before termination flagged the problem as unsatisfiable.
The complete MC-DCOP algorithm was built on top of Adopt, one of the most prominent complete DCOP algorithms [31, 1]. While the implementation was built on the original Adopt algorithm, the same techniques could be applied to more efficient versions of the Adopt algorithm, e.g. BnB-Adopt and Adoptng [45, 41]. The incomplete algorithms built on the k-optimal Maximum Gain Message (MGM) algorithms. K-optimal algorithms are locally optimal algorithms where k indicates the locality of the optimal the algorithm reaches [34]. The techniques detailed in this article could be applied to other complete (OptAPO, SynchBB, NCBB [27, 18, 47, 4]) and incomplete algorithms (DSA, DBA [34, 14, 48, 26]). Proofs of correctness are provided for all of our algorithms.

Experimental results demonstrate that bounded optimization problems are most challenging when resource constraints are tight but satisfiable. In problems where there are insufficient resources, the team goal is largely irrelevant. In problems with ample resources, the local resource constraints require little consideration. The incomplete algorithms are two orders of magnitude more efficient than the complete algorithm for the most challenging problems. The incomplete algorithms are also extremely scalable.

The rest of the article is organized as follows: Section 2 provides a definition of the problem, provides background and motivating domains. Section 3 discusses our complete algorithm as well as proofs and experimental results. Section 4 and 5 presents incomplete algorithms, proofs and experimental results. Section 6 provides related work.

2 Problem Definition

2.1 DCOP Definition

A DCOP[31, 1, 38, 27] consists of n variables, \{x_1, x_2, ..., x_n\}, assigned to a set of agents who control the values they take on. Variable \(x_i\) can take on any value \(d_i\) 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 if 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 \((x_1 \leftarrow 1, x_2 \leftarrow 1, x_3 \leftarrow 1, x_4 \leftarrow 1)\). (We will discuss the g-cost table later). While the above commonly used definition of DCOP emphasizes binary constraints, general DCOP representations may include n-ary constraints.

While we have defined DCOPs above in terms of cost functions, the problem can be equivalently defined in terms of reward functions and maximization of
rewards. Indeed, traditionally research on complete algorithms has focused on cost minimization\cite{31, 1, 38, 27}, whereas incomplete algorithms have often been defined in terms of reward maximization\cite{34, 35}. In accordance with this legacy, and to ensure consistency when discussing previous work, we use the cost minimization approach for complete algorithms, and reward maximization when describing incomplete algorithms. However, these approaches are easily interchangeable.

### 2.2 Motivating Domains

This section demonstrates the need to extend the expressivity of DCOP with examples from two domains: distributed meeting scheduling and distributed software development.

**Distributed Meeting Scheduling:** In these problems, groups of researchers at different locations need to meet. Each researcher has preferences over when and where the meetings he/she is involved in will occur. However, there are costs associated with traveling to meetings. The money for these expenses is allocated from travel budgets maintained by group leaders in each research group. The problem is to maximize the sum of everyone’s satisfaction with the schedule, while ensuring that the travel costs accrued by researchers do not exceed the travel budget for their group. Thus, while the travel budgets impose local resource constraints on a group of agents, we need to simultaneously globally optimize the overall meeting schedule. In addition, while researchers are willing to share their scheduling preferences with each other, the budgetary information needs to be kept private.

The existing DCOP formalism is able to model meeting time preferences using \( f \)-cost functions, however, it cannot handle multiple types of constraints being defined on the same set of variables. In particular, it cannot capture both the global and local aspects of the problem simultaneously. In addition, the travel budgets are private \( n \)-ary constraints, and DCOP cannot explicitly represent individual constraint privacy.

**Distributed Software Development:** Many software companies have campuses around the world, so teams must collaborate across time zones \cite{9, 21}. Interdependent tasks within a project must be scheduled for their timely completion. To facilitate the handoff, a team liaison must videoconference with the team.
receiving the code during the initial stages of development. This liaison may
have to stay after hours for the videoconference. To avoid burnout, corporate
policy may limit liaison overtime. The overtime limits introduce a set of n-ary
satisfaction constraints that are publicly known. Once again, DCOP can cap-
ture the scheduling preference constraints or it could capture the satisfaction
constraints (since they are non-private), however, it cannot express both sets of
constraints simultaneously.

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 additional resource constraints, which may or
may not be publicly known.

2.3 Multiply-Constrained DCOP

To address the expressivity limitations, this article defines Multiply-Constrained
DCOP (MC-DCOP). MC-DCOP adds a new cost function \( g_{ij} \) on a subset of the
links connected to node \( x_i \) and a g-budget \( G_i \) which the accumulated g-cost on
local links must not exceed. Together the g-function and g-budget constitute a
g-constraint. Figure 1 shows an example with g-constraints on \( x_1 \) and \( x_4 \). In the
example, if \( x_1, x_2, x_3 \) each take on the value of 1 (leading to an optimal f-cost)
then \( x_1 \)'s local g-cost is 7, which violates \( x_1 \)'s g-budget of 4. G-constraints may
be either private or shared. The MC-DCOP objective is to find \( A \) s.t. \( F(A) \) is
minimized, where:

\[
F(A) = \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j) \text{ where } x_i \leftarrow d_i, x_j \leftarrow d_j \in A
\]

and

\[
\forall x_i \in V : \sum_{x_i, x_j \in \text{neighbors}(x_i)} g_{ij}(d_i, d_j) \leq G_i
\]

For an illustration of the expressive power of MC-DCOP, consider the meet-
ing scheduling domain mentioned in the previous section. We may use the f-cost
function to model researchers’ meeting preferences using methods such as [24].
In particular, using the PEAV representation of [24] for standard DCOPs, the
domain of each variable is the possible meeting times and locations; the f-cost
expresses agents scheduling preferences and that they must agree on the meet-
ing time (or else receive an infinite cost). The key change with MC-DCOP is
that it now allows us to use g-cost to represent the travel cost incurred by the
group leader, given the values of meeting time and location assigned to differ-
ent variables (i.e. due to the meetings of the different members in the group
leader’s group). We use the g-budget to represent the travel budget of group
leaders, which may need to be private. The goal is then to ensure that the meet-
ing schedules are optimized while simultaneously ensuring that group leaders’
travel budgets (i.e. g-budgets) are not exceeded.

In the motivating domains from Section 2.2, the resource constraints and
optimization constraints represented fundamentally different items: time prefer-
ences and money in the meeting scheduling domain and completion time and
overtime pay in the software development domain. However, using a definition of DCOP that allowed for n-ary constraints, rather than the more common binary constraints, would allow us in theory to combine an agent’s resource and optimization constraints into a single n-ary constraint. An example is shown in Figure 2, where $x_0$’s resource constraint is combined with the optimization constraints on the $x_0 - x_1$, $x_0 - x_2$ and $x_0 - x_3$ links to form the 4-ary constraint shown on the right. Instead of taking this approach, we chose to retain the natural factoring of the two types of constraints because it allowed for greater efficiency and fine-grained control over the privacy-efficiency tradeoff. To understand why using un-factored constraints is less efficient it is worth noting that a common approach to handling n-ary constraints, and the one proposed although not implemented in [31], is to encapsulate the n-ary constraint inside a new variable ($x'$) and place the variable somewhere in the DCOP network where it will receive messages detailing the values chosen by each of the variables in the original constraint ($x_0 \ldots x_3$). In order for $x'$ to evaluate its n-ary constraint, it must wait for $x_0 \ldots x_3$ to select values. However, $x_0 \ldots x_3$ no longer have access to the information about how their value choices affect both the f-cost accrued on each link and the resource expenditure on each link. This prevents them both from using resource information to pre-emptively prune domain values (e.g. the value B can be pruned from $x_1 \ldots x_3$’s domains whenever $x_0$ takes on R) and from using f-cost information to perform opportunistic search (e.g. the value W is more promising for $x_1 \ldots x_3$ to try than the value B, if $x_0$ has selected Y). The inability to use the factored constraint information greatly worsens the performance of an n-ary DCOP algorithm on MC-DCOP problems, so we chose to maintain the natural factoring of the constraints in creating our MC-DCOP algorithms.

<table>
<thead>
<tr>
<th>$x_{di}$</th>
<th>$x_{dj}$</th>
<th>$f(x_{di}, x_{dj})$</th>
<th>$g(x_{di}, x_{dj})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>B</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>R</td>
<td>W</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>B</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>W</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_{di}$</th>
<th>$x_{dj}$</th>
<th>$f(x_{di}, x_{dj})$</th>
<th>$g(x_{di}, x_{dj})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>B</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>R</td>
<td>B</td>
<td>5</td>
<td>$\infty$</td>
</tr>
<tr>
<td>R</td>
<td>W</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>W</td>
<td>5</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Y</td>
<td>B</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>B</td>
<td>5</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Y</td>
<td>W</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>W</td>
<td>5</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Figure 2: Combining Resource and Optimization Constraints
3 Complete MC-DCOP algorithms

Multiply-Constrained Adopt (MCA) is a novel algorithm that builds upon the highly efficient DCOP algorithm Adopt. While our implementation built on the original Adopt algorithm, the same techniques could be applied to newer, more efficient versions of Adopt, e.g., BnB-Adopt and Adopt-ng [45, 41]. Section 3.1 provides background on the Adopt algorithm. Section 3.2 describes the key techniques employed in MCA, Section 3.3 describes MCA in detail, Section 3.4 provides correctness and complexity results and Section 3.5 contains the experimental results.

3.1 Background: ADOPT

Adopt[31] is an asynchronous complete DCOP algorithm. Adopt has been discussed in detail in the literature[31, 1, 5], 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 article, we will use the terms variable and node interchangeably.

Adopt employs two basic messages: VALUE and COST.\footnote{Adopt also uses THRESHOLD messages for improved efficiency, and this message is also used in our algorithms. Since this efficiency enhancement is orthogonal to the main contributions in this paper, we do not discuss the THRESHOLD messages.} Assignments of values to variables are conveyed in VALUE messages that are sent to neighbors 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.

3.2 Basic Ideas

As discussed in Section 1, the three key ideas in MCA are: constraint-graph transformation, dynamically constraining search, and local acyclicity. We de-
scribe these basic ideas first before providing a more detailed description of the algorithm.

3.2.1 Constraint-Graph Transformation

To exploit existing DCOP algorithms and maintain privacy, virtual variables are added to enforce n-ary g-constraints. Each virtual variable is responsible for the enforcement of a single g-constraint. A virtual variable collects information from all of the regular variables involved in its g-constraint and performs a centralized computation to see if the g-constraint is currently violated. If it detects a g-constraint violation, the virtual variable asynchronously preempts the current search path. Using such extra (virtual) variables to enforce n-ary satisfaction constraints via binary satisfaction constraints has appeared in the centralized CSP literature [42]. The use of this idea in MCA has 3 novel aspects.

- These virtual variables are embedded in an existing singly-constrained DCOP algorithm. To ensure the correctness of the resulting asynchronous multiply-constrained DCOP algorithm, Adopt’s DFS tree is restructured and the virtual variables are appropriately placed in the tree.

- Since the constraint-graph restructuring can change local acyclicity properties of other variables, MCA’s preprocessing identifies and preserves local acyclicity where feasible.

- The local g-constraint is encapsulated in a virtual variable which is owned by the same agent as the original variable and placed as a leaf in the DFS tree. This allows the privacy of both the g-function and the g-budget to be protected from other agents in the network because the only information required by or revealed to the other agents is that their current assignment is unacceptable. Virtual variables differ from regular variables in that they have no domain values of their own, they simply send feedback in response to the domain choices of other variables. However, virtual variables are not independent agents, they are controlled by the same agent as the original variable, despite being located in a different part of the communication network. An example virtual variable can be seen in Figure 3a.

3.2.2 Dynamically Constraining Search

To mitigate against the increased complexity of bounded optimization problems, it is important to exploit g-constraint revelation when allowed to gain efficiency in the search for the optimal solution. This is achieved by requiring descendant nodes to only consider assignments that will not violate their ancestors’ g-constraints. Specifically, each descendant is passed a bound (termed g-threshold) specifying how large a g-cost it can pass up, limiting its effective domain. This g-threshold is an exact bound when the local constraint graph is acyclic and an upper bound otherwise. If a potential value assignment fails to satisfy an ancestor’s g-threshold, variables will not explore this value for f-cost.
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.

3.2.3 Local Acyclicity (T-nodes)

The notion of local acyclicity is captured formally by the definition of T-nodes. A 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 its lower priority neighbor $x_3$ is not also its child. However, $x_2$, $x_3$ 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.

MCA exploits the fact that g-constraints are local constraints to allow these three techniques to be applied simultaneously to different variables within the same problem. As a result privacy protecting techniques like constraint-graph transformation apply only at those variables requiring that their g-constraint be kept private. Thus, having a single private g-constraint in the problem does not preclude the other variables from using the more efficient dynamically constraining search and local acyclicity techniques.

3.3 MCA Algorithm Description

Figures 4 and 5 present pseudo-code for MCA. MCA uses three subalgorithms: (i) MCA Private (MCAP) is used when a g-constraint may not be revealed: it uses constraint-graph transformation, (ii) MCA Shared (MCAS) is used when a g-constraint is non-private, but the variable is not a T-node: it employs constraint-graph transformation and dynamically constraining search, and (iii) MCA Shared and Acyclic (MCASA) is utilized when a g-constraint is non-private and the variable is a T-node: it uses dynamically constraining search and local acyclicity. Each sub-algorithm will be discussed separately, and then their interaction in the unified algorithm will be described.

3.3.1 MCAP

In MCAP, privacy is taken to mean that neither the g-constraints nor the variables involved are explicitly revealed to other agents. Since the g-constraints are not visible other agents, MCAP uses Adopt’s normal opportunistic search mechanisms to search for an optimal f-cost solution. To handle the additional g-constraints, MCAP has a set of virtual variables. Each virtual variable is responsible for enforcing the g-constraint of a single variable. An illustrative snapshot of MCAP’s execution is shown in Figure 3a. In the example in Figure 3a, $x'_1$ is a virtual variable that is responsible for enforcing the g-constraint of variable $x_1$. The virtual variable is controlled by the agent that owns the original variable (lines 3-5 in Figure 4), so $x_1$ and $x'_1$ both belong to the same agent.
A virtual variable receives VALUE messages from all of the variables in its particular g-constraint. It looks at the current full or partial assignment and tests whether its g-constraint is violated. If the current assignment violates the g-constraint, the virtual variable sends out a feedback signal, in the form of an infinite COST message (lines 60-63 in Figure 5). The feedback is sent to the lowest priority variable involved in the violated constraint, which in the example in Figure 3a would be $x_3$. 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.

Since feedback must be able to percolate up to all the nodes in a particular g-constraint, Adopt’s DFS tree must be restructured to put all the variables in a particular g-constraint in the same subtree (lines 6-8). MCAP preprocessing then 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). Being a leaf allows the virtual variables to receive VALUE messages from all of the variables in the g-constraint without having to send any itself. The preprocessing of the DFS tree is revisited in Section 3.3.4.

MCAP protects privacy in several ways. Since the g-constraint is encapsulated in the virtual variable, no other agent explicitly needs to know either the g-budget or the g-functions. Other variables just search for the optimal f-cost solution. Additionally, since the feedback sent by the virtual variables is a regular COST message and agents do not know the global graph structure, the fact that a variable is a virtual variable can be obfuscated. The receipt of an infinite COST message from a child can mean one of three indistinguishable things:

- the child is a virtual variable
- the child is a regular variable passing up an infinite cost from a virtual variable lower in the DFS tree
- the child is a regular variable passing up an infinite cost incurred due to the f-cost constraints
3.3.2 MCAS

MCAP’s partial search of unsatisfying assignments slows the algorithm. When a g-constraint does not have to be kept private, the g-function of a link is revealed to both the variable with the g-constraint and the variable on the other end of the link. MCAS and MCASA, shown in Figures 3b and c respectively, exploit g-function revelation by having nodes send their descendants g-thresholds (line 52). These g-thresholds indicate that the descendant may take on no value with a g-cost greater than the g-threshold. The snapshot in Figure 3b shows the g-thresholds being passed down from $x_1$ to $x_2$ and $x_3$, causing some values to be eliminated from the domains of $x_2$ and $x_3$.

If the variable is not a T-node, which is the case in MCAS, the g-threshold for a child ($x_i$) is an upper bound: the total g-budget minus the minimum g-cost that could be consumed by each of the other links:

$$G_i - \sum_{x_j \in \text{Neighbors}(x_i) \neq x_1} \min\{g_{ij}(d_i, d_j)\}$$

Given this bound, a node can prune values from its domain that do not satisfy the g-threshold. This pruning of the domain reduces the search space and leads to speedups over MCAP. Nonetheless, as in MCAP, MCAS still requires the virtual variable to correctly enforce g-constraints as shown in Figure 3b. In particular, since g-threshold messages in MCAS only provide upper-bounds, values taken by multiple children may violate a parent’s g-constraint in the absence of the virtual variable.

3.3.3 MCASA

MCASA is applied when a variable’s g-constraint is not required to be kept private and the variable is a T-Node. For T-nodes, it is possible to calculate the best way to allocate resources between a node’s children. This allocation provides an exact bound on g expenditure which enables more values to be pruned from the domain of each child and further speed-ups to be achieved. Calculating the optimal split requires a node to maintain a mapping from potential g-thresholds to lower bounds on f-costs (GFmap) for each of its children. In essence, the GFmap maintains for each potential value of the g-threshold the minimum f-cost that each child could report — with higher g-thresholds yielding lower f-costs. The GFmap is dynamically initialized from the link f- and g-functions (line 17-18; line 50) and then updated as COST messages arrive (line 38).

2The inspiration for g-thresholds came from the THRESHOLD messages used as an efficiency mechanism in Adopt. In Adopt THRESHOLD messages were used when re-creating a past solution to tell children not to bother looking for an assignment with an f-cost < f-threshold because previous exploration had determined that such an assignment did not exist. G-thresholds, by contrast, do not require any exploration to calculate; they pre-emptively tell the children not to spend more than a specified quantity of g because such an assignment would break the g-constraint.
A T-node uses the g-threshold to f-cost mappings to calculate how to split its remaining g-budget among its children. The remaining g-budget is its total g-budget minus the g-cost consumed on each of the links with its ancestors. The optimal split minimizes the sum of the f-costs that will be reported by the variable’s children, as estimated in the GFmaps. Each time a GFmap is updated the optimal split is recalculated by the calcOptimalSplit function, which is implemented using a centralized dynamic program. As the costs from the subtrees rooted at each child percolate up the tree, the GFmaps will better and better reflect the actual costs of particular allocations of resources. Eventually, the algorithm will settle on the globally optimal split.

Recall from Section 3.1 that a key feature of Adopt is that a node receives lower bounds (lb) and upper bounds (ub) from children and passes up these bounds to the parent node; a modification is required in MCASA when dealing with these bounds. In particular, since \( x \)'s lower bound and upper bound for the current context now depend upon the way \( x \) splits its g-budget among its children, \( x \)'s current split is now part of the context for lb and ub. Variable \( x \) stores the g-budget split as part of its context and when \( x \)'s children send COST messages, they include the g-threshold that was in effect when they sent the message. The lb and ub are reset to 0 and \( \infty \) respectively whenever \( x \) changes its split (line 39) as well as after normal context changes (line 28). If a node cannot find an assignment that satisfies the current g-threshold, it will send up a cost of \( \infty \) to its parent which will cause its ancestors to switch their values (line 45).

While it’s true that MCASA is only applicable in fairly limited situations: strict hierarchies (only parent-child links) and peripheral nodes, it significantly speeds up computation and can be applied on a per-node basis. Thus it is worth including in the suite of algorithms despite its limited applicability.

### 3.3.4 Combining Techniques

The subalgorithms may be applied simultaneously to different nodes within the same problem. The g-constraints are local constraints, so the only challenging situations occur when a parent and a child are using different techniques. The algorithm handles these situations as follows:

- **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 an infinite COST message percolates up from the child’s virtual variable.

- **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 performed as part of the preprocessing for MCAP and MCAS can turn a T-node somewhere else in the tree into a non-T-node which would prevent the use of MCASA. There are
An initial DFS tree is assumed to have been built in preprocessing. 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 $\text{Tnode}_i == \text{false}$ or $\text{private}_i == \text{true}$
3. $x'_i$ is a new virtual variable
4. $\text{Neighbors}(x'_i) \leftarrow \text{Neighbors}(x_i) \cup x_i$
5. $\text{Neighbors}(x_i) \leftarrow \text{Neighbors}(x_i) \cup x'_i$
6. forall $x_k \in \text{Children}(x_i)$
7. if $x_k$ is not a neighbor of $x_l \in \text{Children}(x_i)$
8. $\text{Neighbors}(x_k) \leftarrow \text{Neighbors}(x_k) \cup x_l$
9. rebuildDFStree($x_1 \ldots x_n$)
10. forall virtual variables $x'_i$, $\text{parent}(x'_i) \leftarrow \text{lowest priority Neighbor of } x'_i$

**Initialize**

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

when received $(\text{VALUE}, x_j, d_j, g\text{Thresh}_j)$

24. if $\text{private}_j == \text{false}$
25. add $(x_j, d_j, g\text{Thresh}_j)$ to $\text{CurrentContext}$
26. else
27. add $(x_j, d_j)$ to $\text{CurrentContext}$
28. reset lb and ub if $\text{CurrentContext}$ has changed
29. if $\text{private}_i == \text{false}$ and $\text{Tnode}_i == \text{true}$
30. if $\text{CurrentContext}$ has changed
31. forall $x_l \in \text{Children}$
32. $\text{GFmap}(x_l, gt) \leftarrow \min f(d_l, d_i)$ s.t. $g(d_l, d_i) \leq gt$
33. calcOptimalSplit
34. backTrack;

Figure 4: Multiply-Constrained Adopt Pseudo-code, part 1
when received \((\text{COST}, x_k, \text{context}, \text{lb, ub})\)

(35) update \(\text{CurrentContext}\)

(36) if \(\text{context}\) compatible with \(\text{CurrentContext}\)

and \(T\text{node}_i == \text{true}\) and \(\text{private}_i == \text{false}\)

(37) \(\text{GFmap}(x_i, g\text{Threshold}) \leftarrow \text{lb}\) s.t.

\((x_i, d_i, g\text{Threshold})\) is part of \(\text{context}\) from \(x_k\)

(38) \(\text{calcOptimalSplit}\)

(39) if any \(g\text{Threshold}\) has changed, reset \(\text{lb}\),\(\text{ub}\)

(40) else store \(\text{lb}\),\(\text{ub}\)

(41) else if \(\text{context}\) compatible with \(\text{CurrentContext}\) and \(g\text{Threshold}\)

(42) store \(\text{lb}\) and \(\text{ub}\)

(43) \(\text{backTrack}\)

\begin{verbatim}
procedure \text{backTrack}
(44) if \(x_i\) not a virtual variable
(45) if no \(d\) satisfies \(g\text{Threshold}_j\) \(\text{LB, UB} \leftarrow \infty\)
(46) else if \(\text{LB}(d_i) > \text{LB}(d)\) for some \(d\)
(47) \(d_i \leftarrow d\) that minimizes \(\text{LB}(d)\) and satisfies \(g\text{Threshold}_j\)
(48) if \(T\text{node}_i == \text{true}\) and \(\text{private}_i == \text{false}\)
(49) \(\text{GFmap}(x_i, gt) \leftarrow \min f(d_i, d_i)\) s.t. \(g(d_i, d_i) \leq gt\)
(50) \(\text{calcOptimalSplit}; \text{reset lb and ub}\)
(51) if \(\text{private}_i == \text{false}\)
(52) \(\text{SEND (VALUE, (x_i, d_i, g\text{Threshold}(x_k)))}\)

to each lower priority neighbor \(x_k\)
(53) else \(\text{SEND (VALUE, (x_i, d_i))}\)

to each lower priority neighbor \(x_k\)
(54) if \(\text{LB} == \text{UB}\):
(55) if \(\text{TERMINATE}\) received from parent or \(x_i\) is root:
(56) \(\text{SEND TERMINATE}\) to each child
(57) \(\text{Terminate execution;}\)
(58) \(\text{SEND (COST, x_i, CurrentContext, LB, UB)}\)
(59) else % else a virtual variable
(60) if \(g(d_i, CurrentContext) > g\text{Budget}(x_i)\)
(61) \(\text{SEND (COST, x_i, CurrentContext, } \infty, \infty)\) to parent
(62) else
(63) \(\text{SEND (COST, x_i, CurrentContext, 0, 0)}\) to parent
\end{verbatim}

Figure 5: Multiply-Constrained Adopt Pseudo-code, part 2
two steps in the preprocessing that 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 6a, \( x_1 \) has a g-constraint with \( x_2, x_3 \) and \( x_4 \). During preprocessing, a link must be added between \( x_3 \) and \( x_4 \), because otherwise the tree building algorithm will attempt to place non-neighbors in separate subtrees to increase parallelism. Additionally, one variable 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 (see Figure 6b).

In contrast, in Figure 3b 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. Since \( x'_1 \) need not be given a g-threshold, the fact that it is not one of \( x_2 \)'s children does not prevent \( x_2 \) from being a T-node.

Given these two cases, 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.
3.4 Correctness and Complexity of MCA

In this section the proofs for each subalgorithm are handled separately for the sake of clarity. As previously described, the interaction of the techniques in the combined algorithm does not change their properties. In the following proofs 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 ($\delta$) plus the sum of each of $x_i$’s children’s ($x_l$’s) costs, is minimized while satisfying the $g$-constraint:

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

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

**Proof:** The proof starts from the correctness of the original Adopt algorithm [31]. At every node $x_i$ Adopt will find:

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

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

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

Part (1) uses 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 $f$, variables in MCAP will never take on a final 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. As a result Adopt’s normal mechanisms ensure it will find the minimum $f$-cost solution.

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

**Proposition 2** If the $g$-constraint for each node $x_i$ is:

$$\sum_{x_j \in \text{Neighbors}(x_i)} g_{ij}(d_i, d_j) < G_i$$

then no satisfying solution can contain on link $l_{id}$ a $g$-cost greater than:

$$G_i - \sum_{x_j \in \text{Neighbors}(x_i) \neq x_i} \min\{g_{ij}(d_i, d_j)\}$$
Proof: Each link consumes a certain minimum $g$-cost, and MCAS only subtracts the sum of the absolute minimum costs on all links.

For MCASA, if the $g$-thresholds are assigned optimally, then given the correctness of the original Adopt algorithm, MCASA is correct.

**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:** Assume $x_k \in \text{Children}(x_i)$ and that there are no non-T-nodes as descendants of $x_i$. Additionally, assume MCASA terminates with $g$-thresholds $g'_{ik}$ ($\forall x_k$) which are not optimal. Thus there exists another set of $g$-thresholds ($g^*_{ik}$) such that:

$$\delta(d_i) + \sum_{x_k} \min \{ \text{lb}(d_i, d_k^*) \} < \delta(d_i) + \sum_{x_k} \min \{ \text{lb}(d_i, d'_k) \}$$

where $d_k^* \in \{ D_k | g_{ik}(d_i, d_k) \leq g_{ik} \}$

where $d'_k \in \{ D_k | g_{ik}(d_i, d_k) \leq g'_{ik} \}$

Since local cost $\delta(d_i)$ is constant for all $g$-thresholds, it can be ignored. To have been selected, $g'_{ik}$ must have seemed optimal based on the current information at some point. To distinguish these two states of knowledge let the following terms be defined:

- $f_{\text{actual}}(g_{ik}, x_k)$ is the f-cost (specifically $\min \{ \text{lb}(d_i, x_k) \}$ where $d_k \in \{ D_k | g_{ik}(d_i, d_k) \leq g_{ik} \}$) when all costs have percolated up from all descendants
- $f_{\text{current}}(g_{ik}, x_k)$ is the current lower bound on f-cost

When $g'_{ik}$ is selected:

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

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

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

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

(by 1)

$$\sum_{x_k} f_{\text{current}}(g'_{ik}, x_k) > \sum_{x_k} f_{\text{current}}(g_{ik}^*, x_k)$$

(by 4)
Thus, based upon current information \((f_{\text{current}})\) MCASA will switch from \(g'_{ik}\) to \(g_{ik}^*\) because it has a lower associated \(f\)-cost. This contradicts the 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 \((\delta)\) plus the sum of \(x_i\)'s children's costs, is minimized while satisfying the \(g\)-constraint, as the counter-example in Figure 7 shows.

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(d_1)</th>
<th>(d_2)</th>
<th>(f)</th>
<th>(g)</th>
<th>(g)-budget = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(x_2)</td>
<td>(d_2)</td>
<td>(d_3)</td>
<td>(f)</td>
<td>(g)</td>
<td>(g)-budget = 2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: MCASA fails on non-T-nodes

In Figure 7, \(x_1\) is a non-T-node since \(x_3\) is a child of \(x_2\) not \(x_1\). If it is assumed 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 \(x_1\) (\(x_2\) tries out both values under given threshold of 2). Since increasing \(x_2\)'s \(g\)-threshold does not stop \(x_2\)'s \(g\)-constraint from being violated\(^3\), \(x_1\) 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 \(f_{\text{current}}(gt, x_2) = \infty\) whereas \(f_{\text{actual}}(gt, x_2) \leq 4\) where \(gt \in \{1, 2, 3\}\). □

The final issue is that of MCA's runtime and space complexity. The original DCOP problem is known to be NP-hard. The algorithm for solving the MC-DCOP problem adds at most \(n\) additional nodes, so the runtime complexity class has not worsened. While adding \(n\) extra nodes to the tree does not change the complexity class of the problem, it does significantly increase the runtime, as will be described in Figure 8. This is because MC-DCOP is 'harder' than DCOP in a colloquial sense of the term because the set of constraints in the problem is much larger, so any algorithm that solves an MC-DCOP will run slower, send more messages or use more space than one that solves a regular DCOP. A key

\(^3\)Variable \(x_1\) will not try lowering \(x_2\)'s \(g\)-threshold because logically a lower \(g\)-threshold should only worsen the situation.

\(^4\)This counter example is based on having just a \(g\)-function on the \(x_2 - x_3\) link, but, similar examples can be constructed using just an \(f\)-function or both an \(f\)- and a \(g\)-function on the link.
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 up to \( n \) virtual variables have been added, 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. The addition of the mapping causes the space complexity to be \( |D_i| n + G, n \).

### 3.5 Experimental Results for MCA

This section presents five sets of experiments. The first compares the performance of MCAP, MCAS and MCASA on four settings. Setting 1 comprises 20-node problems, with 3 values per node, an average link density of 1.9 and maximum link density of 4. It has 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 is 85% T-node (which increases the average link density to 2.2) which allows us to evaluate the impact of T-nodes. Settings 3 and 4 are similar to settings 1 and 2 respectively, except that they are 10-node problems. Fifteen sets of constraint functions were created for each of the domain settings, so each data-point in the graphs in this section is an average over 15 problem instances.

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

To highlight the tradeoff between the subalgorithms, Figure 8 shows the performance of each subalgorithm when applied to all the nodes in a problem. 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, which is measured in cycles where one cycle consists of all agents receiving incoming messages, performing local processing and sending outgoing messages[31]. The y-axis is logarithmically scaled. The y-axes are not identical in the two graphs.

The graphs show that MCAP has the poorest performance. This result is caused by its preserving the privacy of g-constraints. 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. 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. MCASA cannot be applied to all the nodes in these settings.) The switch from 100% T-node to 85% T-nodes causes a significant increase in runtime.

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. MCASA cannot be applied to all the nodes in these settings.) The switch from 100% T-node to 85% T-nodes causes a significant increase in runtime.

For all of the subalgorithms, the runtime curves in Figure 8 have a hill shape: the run-times are lowest at high g-budgets, which correspond to no resource constraints, and at low g-budgets, which correspond to tight resource constraints. This suggests that bounded optimization problems are most challenging when the resources are sufficient but not plentiful. The data shown in Figure 9a suggest an explanation. This figure plots the g-budget on the x-axis and the total number of infinite cost messages received by any variable from a virtual variable. The figure shows results from three representative cases from setting 3 when running MCAP. The same hill can be seen appearing at a g-budget of 10 and diminishing to almost 0 at a g-budget of 20. This shape is consistent 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 takes longer to

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 curves in Figure 8 have a hill shape: the run-times are lowest at high g-budgets, which correspond to no resource constraints, and at low g-budgets, which correspond to tight resource constraints. This suggests that bounded optimization problems are most challenging when the resources are sufficient but not plentiful. The data shown in Figure 9a suggest an explanation. This figure plots the g-budget on the x-axis and the total number of infinite cost messages received by any variable from a virtual variable. The figure shows results from three representative cases from setting 3 when running MCAP. The same hill can be seen appearing at a g-budget of 10 and diminishing to almost 0 at a g-budget of 20. This shape is consistent 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 takes longer to
discover unsatisfiability of partial solutions, leading to longer run-times and the hill shape.

In all settings, for low g-budgets, the MCAS algorithm outperforms MCAP. However, for high g-budgets, there is no difference in performance. The narrowing of the performance difference is based on the fact that MCAS prunes fewer domain values when the g-budget is high. In Figure 9b, the g-budget is plotted 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, results are also provided for 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. 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](image)

In real domains, there may be situations in which only some of the agents are concerned about privacy. For example, in meeting scheduling domains, people may trust different individuals or organizations differently with their budgetary information. The experiments in Figure 10 demonstrate the benefits of the per-node application of the different subalgorithms: MCAP, MCAS and MCASA. Here, the examples from settings 1 and 2 from Figure 8a and b were taken and 0%, 25%, 50% and 100% of the nodes were randomly assigned to have private g-constraints while the remaining were assumed to be non-private. In the 25% and 50% cases, all three of the subalgorithms (MCASA, MCAS and MCAP) were executing simultaneously. At 0% private only MCAS and MCASA were executing and at 100% private only MCAP was executing. 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. The key point of this graph is that when we have only a small fraction of the nodes that require privacy (e.g. 25%), it is not worth paying the cost of a fully private (i.e. 100% private) algorithm. Instead, the per node application of the different subalgorithms is very useful as it allows us to gain significant efficiency gains, e.g. at g-budget of 5, for 85% T-nodes, the 25% private case exploits simultaneous execution of our different subalgorithms to run more than an order of magnitude faster than the 100% private algorithm. However, as in Figure 8, when the g-budget on each variable is loose, per-node application of privacy is not very useful — the runtimes converge because no pruning takes place.

4 Incomplete MC-DCOP algorithms

While complete algorithms have the advantage of finding the globally optimal apportionment of scarce resources, their completeness limits their efficiency and scalability. In some domains finding the global optimal is more important than finishing rapidly, while in others the priorities are reversed. To address domains where scalability and efficiency are of primary importance, this section describes a new set of incomplete algorithms for solving bounded optimization DCOPs. Two of the techniques used in the complete algorithms also play a role in the incomplete algorithms: constraint graph transformation and dynamically constraining search.

The following describes k-optimality, which is a way of classifying locally optimal algorithms. It also describes the MGM algorithms, which provide the basis for the locally optimal MC-DCOP algorithms developed as part of this article [26]. In particular, we describe the new algorithms — Multiply-Constrained MGM-1 (MC-MGM-1) and Multiply-Constrained MGM-2 (MC-MGM-2) — which are collectively referred to as the MC-MGM algorithms. The next section then presents experimental results.

4.1 Background

4.1.1 K-Optimality

K-optimality is a way of describing how local an optimal solution an algorithm is designed to reach [26, 35, 34]. It is a useful feature of an incomplete algorithm, because prior research has established two types of theoretical guarantees on the quality of the final solution of k-optimal DCOP algorithms [35, 34]. The first type fixes an upper bound on the number of k-optima that can occur in a problem. The second type establishes a lower bound on the quality of the k-optimum as a percentage of the quality the globally optimal solution. These results allow users to estimate the benefits of using a particular k-optimum
on a particular problem because the efficiency/performance tradeoff has been quantified.

A k-optimal solution is one where no group of k or fewer variables could make a coordinated value change and improve the quality of the overall solution. Figure 11 gives an illustration in which k-optimality $x_1$, $x_2$ and $x_3$ are variables belonging to different agents. The f-reward functions are shown on the two links. In this example, assignments $\{0,0,0\}$ and $\{1,1,1\}$ are 1-optima. The assignment $\{1,1,1\}$ is 1-optimal because variable $x_2$ has no motivation to switch values from 0 to 1 because $x_2$’s local reward would diminish from 16 to 0. Similarly, $x_1$ and $x_3$ have no motivation to switch from 0 to 1 because their local utility would decrease from 5 to 0 and from 11 to 0 respectively. $\{1,1,1\}$ is not a 2-optimal solution because it would be profitable for variables $x_2$ and $x_3$ to make a coordinated move away from the assignment $\{1,1,1\}$. In contrast, $\{0,0,0\}$ is k-optimal for $k = 1, 2$ or 3.

### 4.1.2 MC-K-Optimality

The definition of k-optimality has to be modified when dealing with bounded optimization domains, because the added resource constraints may prevent a set of variables from switching their values to an assignment which would improve the quality of the global solution. This article defines a modification to k-optimality called mc-k-optimality:

- An mc-$k$-optimum is a solution where if any group of $k$ or fewer variables change their values they either a) fail to improve the overall solution quality or b) have at least one g-constraint violation in the resulting assignment.

The example in Figure 12 demonstrates the change. In this example, $x_1$ and $x_2$ are variables belonging to different agents and $x_1$ has a g-budget of 1. The f-reward and g-cost functions are shown on the link between $x_1$ and $x_2$. While $\{1,1\}$ would not be considered a 1-optimum under the traditional definition of 1-optimality, it is an mc-1-optimum, because $x_2$ cannot change its value to the more profitable assignment of 0 due to $x_1$’s g-constraint.
An mc-k-optimal solution may be an unsatisfying solution in the sense that some g-constraints are violated. This can occur for 2 reasons. First, the problem may be a globally unsatisfiable problem, i.e., an assignment such that all g-constraints are satisfied may not exist. The problem in Figure 13a is globally unsatisfiable. None of the four possible assignments satisfies \( x_1 \)'s g-constraint. The second source of unsatisfying mc-k-optima is when the k-optimal algorithm cannot reach a satisfying solution by making local moves. The problem in Figure 13b is not globally unsatisfiable since the assignment \( \{1,1\} \) satisfies \( x_1 \)'s g-constraint. However, in a k-optimal algorithm, the assignments able to be explored are limited by the starting assignment. In hill-climbing algorithms like MGM and Multiply-Constrained MGM, the starting assignments are chosen stochastically. If an mc-1-optimal algorithm were to start out with an initial assignment \( \{0,0\} \), there would be no way for a single variable to change its value and reach a satisfying assignment. This would still be considered terminating at an mc-1-optimum because no single agent can change values and reach a satisfying solution.

4.1.3 MGM-1

This article builds on the 1-optimal algorithm MGM-1 (Maximum Gain Message-1) [26, 34] which is a modification of DBA (Distributed Breakout Algorithm) [48].
In MGM-1, variables begin by initially taking on a random assignment. Then execution continues in rounds. A round is defined as the duration for the system to move from one value assignment to the next. A round could involve multiple messaging phases. Every time a messaging phase occurs, it is counted as one cycle. During a round of MGM-1, each agent broadcasts a gain message to all its neighbors that represents the maximum change in its local utility if it is allowed to act under the current context. An agent is then allowed to act if its gain message is larger than all the gain messages it receives from its neighbors (ties can be broken through variable ordering or another method). Execution continues until no further proposals are made. MGM-1 requires two cycles per round [26, 34]. Note that MGM algorithms, including MGM-1, are assumed to operate in a synchronous fashion; and our newer algorithms that build on MGM algorithms also similarly operate in a synchronous fashion.

We illustrate MGM-1 with an example. Suppose the variables in the example in Figure 11 had initially selected the assignment \( \{0, 1, 0\} \) then \( x_1 \) would send a gain message to \( x_2 \) indicating it could switch values from 0 to 1 and achieve a gain of 5. \( x_3 \) would also send a gain message to \( x_2 \) indicating it could change its value and achieve a gain of 11. \( x_2 \) would send a proposal message to both \( x_1 \) and \( x_3 \) indicating it could switch values and achieve a gain of 30. Since \( x_2 \) has the highest gain, it will switch its value to 0 and the other two variables will remain at their current assignment.

### 4.1.4 MGM-2

With a 1-optimal algorithm, the evolution of the assignments will terminate at a 1-optimum. One method to improve the solution quality is for agents to coordinate actions with their neighbors. This allows the algorithm to break out of some local optima. This section introduces the 2-optimal algorithm MGM-2 (Maximum Gain Message-2) [34].

As with MGM-1, agents initially take on a random assignment and then begin executing rounds of the MGM-2 algorithm. MGM-2 uses randomization to decide which subset of agents are allowed to make offers. Each agent is randomly assigned to be an offerer or a receiver. Each offerer will choose a neighbor at random and send it an offer message which consists of all coordinated moves between the offerer and receiver that will yield a gain in local utility to the offerer under the current context. The offer message will contain both the suggested values for each agent and the offerer’s local utility gain for each value pair. For example, suppose the agents in the example in Figure 11 had currently taken on the assignment \( \{1, 1, 0\} \) and \( x_1 \) had been assigned to be an offerer, while \( x_2 \) and \( x_3 \) had been made receivers. Agent \( x_1 \) could send a proposal to \( x_2 \) that they change values from \( \{x_1 \leftarrow 1, x_2 \leftarrow 1\} \) to \( \{x_1 \leftarrow 0, x_2 \leftarrow 0\} \) for a gain of 5 from \( x_1 \)’s perspective. Given that \( x_3 \) is not a neighbor of any offerers, it will not receive an offer in this round.

After it gets an offer, each receiver calculates the overall utility gain for each value pair in the offer message by adding the offerer’s local utility gain to its own utility change under the new context and subtracting the difference in the
link between the two so it is not counted twice. In the example in Figure 11, Agent $x_2$ would receive the offer and calculate that their combined gain from this move would be 25. If the maximum overall gain is positive, the receiver will send an accept message to the offerer and both the offerer and receiver are considered to be committed. Otherwise, it sends a reject message to the offerer, and neither agent is committed.

Uncommitted agents choose their best local utility gain for a unilateral move and send a proposal message. Uncommitted agents follow the same procedure as in MGM-1, where they modify their value if their gain message was larger than all the gain messages they received. Committed agents send the global gain for their coordinated move. Committed agents send their partners a confirm message if all the gain messages they received from their neighbors were less than the gain for the coordinated move they plan to make. They send a deconfirm message, otherwise. A committed agent will only modify its value if it receives a go message from its partner. MGM-2 requires five cycles (value, offer, accept/reject, gain, confirm/deconfirm) per round in contrast to MGM-1’s 2 cycles per round.

4.2 Multiply Constrained MGM

As mentioned earlier, there are four main challenges that must be addressed in designing locally optimal MC-DCOP algorithms: search complexity, harnessing existing algorithms, privacy/efficiency, and unsatisfiability detection. In addressing these challenges, we once again make use of the constraint-graph transformation and dynamically-constraining search techniques mentioned earlier. Additionally, the unsatisfiability detection challenge is only relevant to incomplete algorithms because incomplete algorithms do not systematically consider all possible assignments and thus cannot easily detect unsatisfiability. There are two approaches to this challenge: a) require a valid initial starting point and maintain a satisfaction invariant (i.e. avoid any over-expenditure of resources) or b) detect when the search has covered all assignments without any being found that are satisfying, but not maintain the satisfaction invariant. This article uses approach a), and add a carefully defined dummy value to variables’ domains so that they can easily find a valid start point and also flag resource constraint violations that remain even at termination. We chose this approach rather than the alternative because it always terminates at a feasible ‘satisfying’ solution and because it reduces the search complexity by ignoring infeasible solutions even if this limits the number of feasible solutions it can reach. It thus uses the additional resource constraints to prune rather than expand the search space.

4.3 MC-MGM-1

MC-MGM-1 may be thought of as containing two separate subalgorithms that can operate simultaneously on the same problem. The first is the shared MC-MGM-1 subalgorithm that makes g-constraint information available to neigh-
MC-MGM-1 \((all\text{Neighbors}, currentValue)\)

1. if \!private
2. SendValueMessages(all\text{Neighbors}, currentValue, available-g-budget)
3. else
4. SendValueMessages(all\text{Neighbors}, currentValue)
5. currentContext = GetValueMessages(all\text{Neighbors})
6. for all newValue in EffectiveDomain(currentContext)
7. \([gain, newValue] = \text{BestUnilateralGain}(currentContext)\)
8. if \(gain > 0\)
9. SendGainMessage(all\text{Neighbors}, gain, newValue)
10. neighborGains = ReceiveGainMessages(all\text{Neighbors}, NeighborValues)
11. if GConstraintViolated(newContext) and !virtual
12. \(n = \text{SelectNeighborsToBlock()}\)
13. SendBlockMessages(n)
14. else if GConstraintViolated(newContext) and virtual
15. \(n = \text{SelectNeighborsToBlock()}\)
16. SendNogoodMessages(n)
17. if ReceivedNogoodMessage()
18. \(NoGoods = NoGoods + [newValue, newContext]\)
19. else if \(gain > \text{max}(neighborGains)\) and !ReceivedBlockMessage()
20. currentValue = newValue

Figure 14: Multiply-Constrained MGM-1 Pseudo-code

boring variables (\textit{dynamically constraining search}). The second is the private subalgorithm which encapsulates g-constraint information in virtual variables (\textit{constraint graph transformation}). The pseudo-code for the combined MC-MGM-1 is shown in Figure 14.

4.3.1 Shared MC-MGM-1

MC-MGM-1 maintains an invariant that at no point during execution does the current assignment violate any agent’s g-constraint. All moves are calculated to go from one assignment where this invariant holds to another assignment where it holds. However, the challenging part is finding the first assignment where this holds. In order to address this issue as well as to provide a mechanism for determining when a problem is unsatisfiable, MC-MGM-1 performs an initialization step when it first begins where it adds a dummy value to each variable’s domain. This dummy value is used as the starting value for each variable. The dummy value, \(d'\), is defined to have the following constraint function on all links:

\[
\begin{align*}
    f(d', d') &= c \\
    g(d', d') &= 0 \\
    f(d', d_i) &= f(d_i, d') = k
\end{align*}
\]
\[ g(d', d_i) = 0 \]

- \( g(d_i) = g(d_i, d') = 0 \)
- \( c \) and \( k \) are constants such that \( c < k < 0 \)
- \( d_i \) is a regular domain value

This definition ensures that all variables can start at an assignment that spends 0 g, which is a satisfying solution. However, since the quality in terms of \( f \) is by definition lower than any real assignment, MC-MGM-1 will attempt to find a solution that does not involve dummy values. The reason that \( c \) must be smaller than \( k \) is that since MC-MGM-1 only allows one variable to move at a time, it must be profitable for nodes to move from a dummy value to a real value even if their neighbors are all still set to their dummy values. Otherwise the initial assignment becomes an mc-1-optimum and termination occurs immediately. If MC-MGM-1 fails to find a valid assignment containing no dummy values then the problem is 1-optimally unsatisfiable, since MC-MGM-1 will always favor real domain values over the dummy one. So the dummy values allow for easy detection of 1-optimal unsatisfiability. It is worth noting that, as mentioned in section 4.1.2, a 1-optimally unsatisfiable problem may be satisfiable in the sense that a solution exists where all g-constraints are satisfied. The problem can still be 1-optimally unsatisfiable if an algorithm cannot reach that satisfying solution in a set of steps where only one agent changes value at a time.

After initialization, each variable repeatedly runs rounds of the pseudo-code shown in Algorithm 14. The first thing the variables do is send value messages to their neighbors informing them of their current value as well as how much g-budget is currently available for use by that particular neighbor. The available-g sent to node \( x_j \) equals total g minus the g currently consumed by all of \( x_i \)'s other neighbors (line 2 in Algorithm 14). This is similar to the available-g sent out during MCAS and, like with MCAS, this can lead to overspending if multiple variables try to use the full available-g.

After receiving the value messages, each node calculates its effective domain (line 6). This means removing from consideration any values that given the current context would violate either the variable’s own g-constraint or any of its neighbors’ available-g’s. The node then considers all of the values in the effective domain and picks the one that would allow it to gain the largest increase in local f, if selected (lines 6-7). It then sends a gain message to all of its neighbors proposing the move and listing the gain that it would achieve (lines 8-9).

Upon receipt of gain messages, two things occur. First the variable looks to see if any of its neighbors can achieve a better gain and if so rescinds its intention to move (line 21). Second, each variable checks to see whether the combined expenditure from all of its neighbors’ proposed moves violates its g-constraint (line 11).

An example of this situation is shown in Figure 15. Variable \( x_1 \) has taken on the value 0 and receives two move proposals from its neighbors \( x_2 \) and \( x_3 \). The neighbors are assumed to currently have taken on the dummy value, thus each sees an available-g of 3. (Notice that in this case, higher g-cost affords us higher f-reward.) They make the following move proposals, neither of which
individually violate the available-g: $x_2$ proposes taking on 0 and $x_3$ proposes taking on 1. However, if both moves are made, $x_1$’s g-constraint will be violated. If a node detects this situation, then it will send a blocking message to a subset of the offending neighbors (lines 12-13). For example, $x_1$ may choose to send a blocking message to $x_3$. (The heuristics for selecting neighbors will be discussed in section 4.3.3.) If a variable receives a block message, then it will not move in the current round. The blocks are temporary and thus an agent is free to consider moving in the next round. Those agents with the highest local gain in the current round who do not receive blocking messages will move. In the example from Figure 15, $x_3$ will not change from the dummy value, but $x_2$ will go ahead and take on the value 0. The available-g will then be updated and $x_3$ will be informed in the next round that it can only propose moves that spend no more than 1 unit of g. These rounds repeat until all agents have ceased to propose moves.

4.3.2 Private MC-MGM-1

The private MC-MGM-1 subalgorithm works like the shared version with two modifications. First, variables cannot explicitly take into account their private neighbors’ g-constraints when proposing moves and second, a private variable’s g-constraint is encapsulated in a virtual variable. Note that the virtual variable’s functionality could be incorporated into the original variable itself. The reason for this is that unlike in MCA, there is no priority ordering among neighbors, so the original variable will receive all the necessary messages to act as its own virtual variable. However, for simplicity of explanation, this article will treat the virtual variable as if it is a separate entity from the original variable.

Initialization in private MC-MGM-1 is the same as that of shared MC-MGM-1 except that for each variable, $x_i$, that has a private g-constraint, a virtual variable, $x'_i$ is created and connected both to $x_i$ and all of $x_i$’s neighbors. The virtual variable $x'_i$ has no domain and its links to other variables have no f-reward or g-cost functions on them, they are purely used for communication.

All of the non-virtual variables select their most profitable potential value, just like in shared MC-MGM-1. However, in this case, the EffectiveDomain() function will not weed out values that would overspend a neighbor’s g-budget (lines 4 and 6), if that neighbor has a private g-constraint. One other difference
is that EffectiveDomain() will consult a list called Nogoods (i.e. permanently blocked values), which lists values for which a Nogood message was received and the context during which it was received. If the current context is the same, then the value from the Nogood list will be eliminated from the effective domain (line 6). These Nogood messages are received from virtual variables and unlike the regular blocking messages of shared MC-MGM-1, they are maintained throughout execution of the algorithm, thus making them a permanent block (lines 18-19). They are made permanent because agents have no other way to know what value assignments to avoid because their neighbors’ g-constraints are private. If a node $x_i$ receives a block message then it will note its neighbors’ values as the context and avoid the specified value until one of its neighbors’ values changes. Variable $x_i$ need only consider its own context, it does not need to worry about the values of variables to which it is not directly connected because if a change in $x_i$’s context is not enough to make the domain choice feasible, it will receive another Nogood message in the new context.

The non-virtual variables send out their proposed moves to all of their neighbors (lines 11-12). At this point the virtual variables use the information from the proposal messages to evaluate whether the g-constraint of the variable they represent will be violated. If it will, then they send Nogood messages to some subset of the variables involved in the g-constraint violation (lines 14-17). If a Nogood message is received by a node then it is added to the Nogoods list along with the appropriate context (lines 18-19). If not, then nodes make their moves just as if they were running shared MC-MGM-1.

### 4.3.3 Heuristics in MC-MGM

Under certain circumstances a variable $x_i$ may receive proposed moves from multiple neighbors such that while no individual move violates $x_i$’s g-constraint, the combined set of moves violates $x_i$’s g-constraint. In this case $x_i$ must send one or more blocking messages. The number of blocking messages sent is the minimum number that will prevent $x_i$’s g-constraint from being violated, which will be at worst one fewer than the number of $x_i$’s neighbors proposing moves (since each individual move is legal). Picking the optimal neighbor to block only using local information is impossible, but there are several possible heuristics for selecting which neighbor(s) to block. Heuristics can be deterministic or stochastic. Deterministic heuristics have the advantage of not ignoring local information in deciding who to block. However, local information may not indicate the globally optimal choice, and so stochastic heuristics have the advantage, when run multiple times, of being able to eventually find the optimal set of neighbors to block. Additionally, there are two ways to handle a blocking message: 1) $x_j$
maintains its old value and chooses not to make its proposed move (monotonic) 2) \( x_j \) resets itself to a value that consumes less \( g \) (non-monotonic). Monotonic heuristics are guaranteed to terminate, but non-monotonic heuristics provide more options for breaking out of a local optimum. For this article, four different heuristics were selected as representative examples of possible heuristics. While examples can be created that cause particular problems for an individual heuristic, experimental results (see Figure 19) demonstrate that on randomly generated examples, the different heuristics produce similar quality results. The four heuristics used are as follows:

- **Monotonic**: \( x_i \) selects one or more random neighbors and sends blocking messages. The blocking messages are interpreted by each neighbor \( x_j \) to mean that \( x_j \) should refrain from changing its value in the current round. The advantage of this heuristic is that it maintains the property of monotonicity which was a property of the original MGM algorithms. The global utility never decreases during execution which allows proof of termination to be guaranteed. However, experimentally, this heuristic is the worst performing of the heuristics on random examples.

- **Random Reset**: \( x_i \) selects one or more random neighbors and sends blocking messages which are interpreted by each neighbor \( x_j \) to mean that it should reset its value to the dummy value. This heuristic, when run multiple times, allows MC-MGM to eventually send its blocking message(s) to the optimal neighbor(s). The disadvantages are that monotonicity cannot be guaranteed and that random reset will not consider changing its own value to prevent the violation.

- **Self**: \( x_i \) sends no blocking messages but instead resets itself to the dummy value. In this case, one fewer cycle of communication is required. However, monotonicity is not guaranteed. This is also a deterministic heuristic.

- **Biggest Spender**: \( x_i \) selects the neighbor \( x_j \) that is using the greatest amount of \( x_i \)’s \( g \)-budget and sends a blocking message which is interpreted by \( x_j \) to mean that it should reset its value to the dummy value. This heuristic attempts to avoid the pitfall of random neighbor selection by choosing a neighbor whose resetting will free up the greatest amount of \( g \). However, it has the disadvantages of a deterministic heuristic.

### 4.4 MC-MGM-2

MC-MGM-2 is also divided into two subalgorithms which can be run simultaneously on different parts of the same problem: Shared MC-MGM-2 and Private MC-MGM-2. The pseudo-code appears in Figure 16.

#### 4.4.1 Shared MC-MGM-2

Multiply-Constrained MGM-2 operates much like MC-MGM-1 with the exception that in addition to making individual moves, it can propose joint moves
MC-MGM-2 \((allNeighbors, currentValue)\)

\(\text{if } !private\)
\(\text{SendValueMessages}(allNeighbors, currentValue, available-g-budget)\)
\(\text{else}\)
\(\text{SendValueMessages}(allNeighbors, currentValue)\)
\(\text{currentContext} = \text{GetValueMessages}(allNeighbors); \text{committed} = false\)
\(\text{if Random}(0,1) < \text{offererThreshold}\)
\(\text{committed} = true; \text{partner} = \text{RandomNeighbor}(allNeighbors)\)
\(\text{for all } newValue \text{ in EffectiveDomain}(currentContext)\)
\(\text{SendOfferMsg}(\text{partner}, \text{bestCoordinatedMove}(\text{partner}))\)
\(\text{for all } newValue \text{ in EffectiveDomain}(currentContext)\)
\([gain, newValue] = \text{BestUnilateralGain}(currentContext)\)
\(\text{offers} = \text{ReceiveOffers}(allNeighbors); \text{offerReplySet} = \cup \text{offers.neighbor}\)
\(\text{if } !\text{committed}\)
\(\text{best} = \text{FindBestOffer(offers)}\)
\(\text{if best.gain} > \text{gain and } newValue \in \text{EffectiveDomain}(currentContext)\)
\(\text{committed} = true; \text{newValue} = \text{best.myValue}\)
\(\text{SendOfferReplyMsg}(\text{partner, commit, best.partnerNewValue, best.gain})\)
\(\text{for all } \text{neighbor} \in \text{offerReplySet}\)
\(\text{SendOfferReplyMsg(neighbor, noCommit})\)
\(\text{if } \text{committed}\)
\(\text{reply} = \text{ReceiveOfferReplyMsg(partner)}\)
\(\text{if reply} = \text{commit}\)
\(\text{newValue} = \text{reply.myNewValue}; \text{gain} = \text{reply.gain}\)
\(\text{else}\)
\(\text{committed} = false\)
\(\text{SendGainMsg}(allNeighbors, gain)\)
\(\text{neighborGains} = \text{ReceiveGainMsgs}(allNeighbors); \text{changeValue} = \text{no}\)
\(\text{if } \text{GConstraintViolated(newContext) and } !\text{virtual}\)
\(n = \text{SelectNeighborsToBlock}()\)
\(\text{SendBlockMessages}(n)\)
\(\text{else if } \text{GConstraintViolated(newContext) and } \text{virtual}\)
\(n = \text{SelectNeighborsToBlock}()\)
\(\text{SendNogoodMessages}(n)\)
\(\text{if } \text{ReceivedNogoodMessage}()\)
\(\text{NoGoods} = \text{NoGoods} + [\text{newValue, newContext}]\)
\(\text{else if } \text{committed}\)
\(\text{if } \text{gain} > \text{max(neighborGains)} \text{ and } !\text{ReceivedgoodMessage}()\)
\(\text{SendConfirmMsg(partner, go)}\)
\(\text{else}\)
\(\text{SendConfirmMessage(partner, noGo)}\)
\(\text{confirmed} = \text{ReceiveConfirmMsg(partner)}\)
\(\text{if confirmed}\)
\(\text{changeValue} = \text{yes}\)
\(\text{else if } \text{gain} > \text{max(neighborGains)} \text{ and } !\text{ReceivedgoodMessage}()\)
\(\text{changeValue} = \text{yes}\)
\(\text{if } \text{changeValue} = \text{yes}\)
\(\text{currentValue} = \text{newValue}\)

Figure 16: Multiply-Constrained MGM-2 Pseudo-code
between pairs of agents. At the beginning of the round agents are stochastically
designated to be either offerers or receivers, this designation helps reduce re-
dundant computation where multiple agents propose the same move (lines 6-7
in Figure 16). If a node, $x_i$, is an offerer, it will randomly select a neighbor,
$x_j$ and search for the best joint move that does not violate $x_i$’s g-constraint or
the available-g of any of $x_i$’s neighbors including $x_j$ (line 8). Variable $x_i$
will then send a proposal message to $x_j$ (line 9). As with regular MGM-2, $x_j$
will not accept a proposal if it is itself an offerer (line 13). However, those receivers
that receive a proposal for a joint move will check to see if it violates their
g-constraint or the available-g of any of their neighbors. If the proposed move
is legal and has the highest local gain, then the node will send an acceptance
message (line 17).

4.4.2 Private MC-MGM-2

Private MC-MGM-2 also uses virtual variables to encapsulate private g-constraints.
The only difference between the private versions of MC-MGM-1 and MC-MGM-
2 is that when blocking a joint move the virtual variable in MC-MGM-2 sends
a nogood message to both nodes planning to move (lines 32-33).

4.5 Proofs

This section presents proofs that when using the monotonic heuristic, the MC-
MGM algorithms are monotonic and terminate in mc-k-optimal solutions. Sim-
ilarly this section gives proofs that when using the monotonic heuristic, MC-
MGM-2 is monotonic and terminates at an mc-2-optimal solution. Many heuris-
tics are available for use within MC-MGM. The proofs in this section depend on
the use of the monotonic heuristic because monotonicity is important in proving
termination. The other three heuristics described earlier were experimentally
found always to terminate, but they have not been proven to terminate. The
following definitions and assumptions are used in the proofs:

- Variables are denoted as $x_i \in X$, where $X$ denotes the set of all variables;
  values of variables are denoted as $d_i \in D_i$ where $D_i$ is the finite domain
  of the variable $x_i$.

- $d^{(n)}$ refers to the assignment of values to variables in the DCOP due to
  MC-MGM at the beginning of the n-th cycle of the algorithm. The global
  utility of this assignment is denoted $U(d^{(n)})$.

- $L(d_i; d_{-i})$ refers to the local utility of agent i, given the current context
denoted as $d_{-i}$. $L(d_i; d_{-i}) = \sum_{x_j \in \text{Neighbors}(x_i)} f_{ij}(d_i, d_j)$.

- $Gain_i$ is the change in local utility of $x_i$ due to a unilateral change in its
  value from $d_i$ to $d'_i$, given fixed context $d_{-i}$, i.e. the difference between
  $L(d_i; d_{-i})$, and $L(d'_i; d_{-i})$.
• Gain_{ij} is the change in local utility of x_i and x_j due to the 2-coordinated change in values from d_i to d'_i and d_j to d'_j. This equals L(d'_i; d_{-i}) + L(d'_j; d_{-j}) + f(d_i, d_j) - f(d'_i, d'_j).

• In computing Gain_i, x_i only considers its EffectiveDomain, i.e., values that do not violate its own g-constraint, or the available-g of those neighbors whose g-constraints are public. Additionally, calculating EffectiveDomain removes from consideration those values that have received nogood messages for the current context.

• Once an agent in MC-MGM takes on a value from its real domain, it will never go back to its dummy starting value, because the gain will be negative.

• MC-MGM maintains as an invariant that no agents' g-constraint is violated at the end of any round of execution.

Proposition 4 In MC-MGM-1, the global utility $U(d^{(n)})$ never decreases.

Proof: There are two separate cases to handle: non-blocking and blocking. In the non-blocking case, no block messages or nogood messages are issued. In the blocking case at least one blocking or nogood message is issued.

In the non-blocking case, if $x_i$ intends to modify its value in round $r$, then:

- $Gain_i > Gain_j$, $\forall x_j \in \text{Neighbors}(x_i)$
- $Gain_i > 0$

Since $x_i$’s neighbors would have received $x_i$’s message proposing $Gain_i$, they will not modify their values in round $r$. Thus, no two neighboring variables will change values simultaneously. So, when $x_i$ changes its value, $Gain_i$ will be realized. Since $x_i$’s gain is the sum of utilities on each link connected to $x_i$, $x_i$’s gain implies that the global utility $U(d^{(r)})$ increases. If multiple variables change values simultaneously, they are guaranteed to be non-neighbors, and thus, each of their gains will add to $U(d^{(r)})$.

In the blocking case, a blocking or nogood message within a round $r$ will cause a variable $x_i$ which had intended to change its value not to make the change. Such a block would cause $x_i$ to realize a gain of 0, which does not cause a decrease in $U(d^{(r)})$. The message does not affect any other variables.

Proposition 5 In MC-MGM-2, the global utility $U(d^{(n)})$ never decreases.

Proof: Variables in MC-MGM-2 can either make individual moves or coordinated moves. The individual moves are equivalent to moves in MC-MGM-1 and have been proven not to decrease $U(d^{(r)})$. This proof will examine the effects of the coordinated moves in both blocking and non-blocking cases.

In the non-blocking case, if $x_i$ and $x_j$ are committed to making a coordinated value change in round $r$, then
• \( \text{Gain}_{ij} > \text{Gain}_{kl}, \forall x_k \in \{\text{Neighbors}(x_i) \cup \text{Neighbors}(x_j) - x_i - x_j\}, \)
  \( x_i \in \text{Neighbors}(x_k) \)
• \( \text{Gain}_{ij} > \text{Gain}_{k}, \forall x_k \in \{\text{Neighbors}(x_i) \cup \text{Neighbors}(x_j)\} \)
• \( \text{Gain}_{ij} > 0 \)

Since \( x_i \) and \( x_j \)'s neighbors would have received \( x_i \) and \( x_j \)'s messages proposing \( \text{Gain}_{ij} \), they will not modify their values in round \( r \). Thus, no two neighboring variables will change values simultaneously unless part of a coordinated move. Thus when the coordinated value change is made, \( \text{Gain}_{ij} \) will be realized. Since the coordinated gain is amassed from the sum of utilities on each link connected to \( x_i \) or \( x_j \), the coordinated local gain implies that the global utility \( U(d^{(r)}) \) increases. If other variables change values during round \( r \), they are guaranteed not to be neighbors of either \( x_i \) or \( x_j \), and thus, each of their gains will add to \( U(d^{(r)}) \). As was the case in MC-MGM-1, the block case would cause the pair of variables to realize a gain of 0, which does not cause a decrease in \( U(d^{(r)}) \).

\[ \square \]

Figure 17: A Deadlock example for MC-MGM-1

Given that MC-MGM sends out blocking messages, there is a possibility of entering deadlock. A cycle of blocking messages could block all variables from changing values, even though they had not yet reached an mc-k-optimum. Figure 17 shows an MC-DCOP where it is possible to enter deadlock. The \( f \) rewards and \( g \) costs are shown in the tables for two of the links; the remaining \( f \) rewards and \( g \) costs are similar. The DCOP is initialized with all agents taking on a dummy value of 0. \( x_1 \) and \( x_3 \) switch from 0 to value R. At this point, \( x_2 \) and \( x_4 \) propose switching to Y, which gives them each a gain of 20, and Y is under the available-g of 2 (for both \( x_1 \) and \( x_3 \)). Since \( x_1 \) and \( x_3 \)'s budgets would be violated if both \( x_2 \) and \( x_4 \) switched to the value Y, they must send blocking messages. If \( x_1 \) randomly selected \( x_2 \) to block and \( x_3 \) randomly selected \( x_4 \) neither \( x_2 \) or \( x_4 \) could change values, and this would create a deadlock situation.

In the monotonic heuristic, the agents being blocked are randomly selected. As a result, remaining in deadlock indefinitely is impossible. Suppose agents can enter deadlock with probability \( p \), where \( p \in [0, 1) \). In Figure 17, \( p = 0.5 \) because there are four ways the two blocking messages could be sent out and 2 result in deadlock. Since agents randomly select who to block each round, there
is a probability of $1 - p$ of escaping deadlock in every round. After $N$ rounds of execution, the probability of remaining in deadlock is $p^N$. Since execution continues until there are no longer any proposal messages being sent, $N$ approaches $\infty$ and $p^N$ approaches 0. Furthermore, once one variable is allowed to change its value, the budgets available at the remaining variables change and the old deadlock is resolved. For example in Figure 17, if $x_2$ is allowed to change its value to Y, then the available-g at $x_1$ and $x_3$ changes to 0, and $x_4$ will propose taking the value P in the next round. A similar example of deadlock can be constructed for MC-MGM-2. While the theoretical guarantee of escaping deadlock relies on the assumption that execution continues for an infinite number of cycles, as a practical matter escaping deadlock usually occurs within a couple of cycles and in no way reduces MC-MGM’s efficiency advantage over complete algorithms.

**Proposition 6** Given that deadlocks are resolved using randomization, MC-MGM-1 will terminate at an mc-1-optimal solution.

**Proof:**
In Proposition 4, it was shown that MC-MGM-1 will lead to a monotonically increasing global utility $U(d(n))$. Since $U(d(n))$ cannot be higher than the finite globally optimal solution, MC-MGM-1 cannot keep increasing $U(d(n))$ forever. Thus, assuming it eventually resolves any deadlocks it enters, MC-MGM-1 will terminate.

Termination in MC-MGM-1 occurs when no variable $x_i$ is able to propose a move from $d_i$ to $d'_i$ given $d_{-i}$ where $Gain_i > 0$ and no g-constraints are violated after the move. This situation is the definition of an mc-1-optimal, so when MC-MGM-1 terminates, the agents have reached an mc-1-optimal.

**Proposition 7** Given that deadlocks are resolved using randomization, MC-MGM-2 will terminate at an mc-2-optimal solution.

Proof is similar to that of the previous proposition.

5 Experimental Results for MC-MGM

This section presents five sets of experiments. For the first four sets of experiments, the testcase suite described in Section 3 is employed. The final experiment focuses on scale-up, and hence a new large-scale testcase suite was used. In contrast with MCA, MC-MGM involves stochastic elements, specifically from three sources: (i) when potential moves have identical gains, which is true at initialization, the variable that moves is chosen stochastically; (ii) when variables in MC-MGM-2 are selecting whether to be offerers and who to make an offer to, the decisions are made stochastically; (iii) when variables are sending blocking or nogood messages, 3 of the 4 heuristics use randomization. Due to
these sources of randomness, each testcase was run 100 times. The results from the 100 runs are averaged together in the graphs presented below.

This first set of experiments demonstrate the runtime savings gained by using the incomplete MC-MGM algorithms. The Random Reset heuristic described in Section 4.3.3 was used with both algorithms for the results presented in this graph because, as is demonstrated in the next Figure, it slightly outperforms the other heuristics that were tested. The examples from settings 2 and 4 from Figure 8 were taken and MC-MGM-1 and MC-MGM-2 (both shared and private versions) were run on the problems. The results are shown in Figure 18a and b. The x-axis shows the g-budget applied to each node and the y-axis measures the runtime in cycles. The runtime represents the time to convergence not to termination because the MGM algorithms on which MC-MGM is built do not have termination detection built in. However, Wellman et al showed that it is possible to implement termination detection in distributed optimization algorithms [44]. Each data point is an average over 100 runs of each of the 15 instances of the problem.

We can see that whereas the runtime for MCA was on the order of hundreds to thousands of cycles, MC-MGM takes only tens of cycles to run. As with MCA, the runtime peaks at a g-budget of about 10 to 15 because that is where the most complicated tradeoff between f and g is taking place. Additionally, the graphs demonstrate that the private versions of MC-MGM-1 and 2 are slower than their non-private counterparts. This is because in the private algorithms the nodes have no knowledge of how their moves will impact their neighbors’ g-constraints and so they expend cycles proposing moves that are then rejected by a virtual variable.

Under certain circumstances an agent $x_i$ may receive proposed moves from multiple neighbors such that while no individual move violates $x_i$’s g-constraint, the combined set of moves violates $x_i$’s g-constraint. In this case $x_i$ must send
a blocking message. There are various possible heuristics for selecting which neighbor to block, four of which were described in Section 4.3.3. In this graph the four heuristics are used in conjunction with MC-MGM-1 on the problems from setting 2 and the quality of the final solution is compared. The x-axis again shows the g-budget at each node and the y-axis measures the global reward achieved. Each data point is an average of the results from 100 runs of each of the 15 testcases in setting 2. Note that in contrast to MCA, these results are expressed in terms of reward (not cost) so a higher final quality is better. As can be seen, in these randomly generated cases, there is very little difference between the quality of the final solutions. For all the other graphs in this section, the Random Reset heuristic was used since it equaled or slightly out-performed the other heuristics.

Figure 19: quality comparisons for different heuristics in MC-MGM-1

Figure 20 shows the difference between the quality of solution obtained using MC-MGM-1 and MC-MGM-2. Once again, problems from setting 2 were used.
and the x-axis measures the g-budget of each variable and the y-axis shows the average global reward of the final solutions. As can be seen, MC-MGM-2 finds a higher quality solution on average than MC-MGM-1 because it is able to make coordinated moves.

Figure 21: Runtime for MC-MGM-2 with varying percentages of private nodes

The next set of experiments demonstrate the advantages of the per-node application of privacy in MC-MGM-2. Here, the examples from setting 2 were taken and 0%, 25%, 50%, 75% and 100% of the nodes were randomly assigned to have private g-constraints while the remaining were assumed to be non-private. The results are shown in Figure 21. The x-axis again shows the g-budget applied and the y-axis measures the runtime in cycles. Each bar in the graph shows an average over 100 runs of the 15 instances. We can see that as the percentage of nodes whose g-constraint is private increases, the runtime increases. Interestingly, the increase in run-time is not uniformly 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 for some cases. Thus even when using an incomplete algorithm it is useful to have fine-grained control over privacy.

The final set of experiments illustrate the ability of our incomplete algorithms to scale-up by running MC-MGM-1 and MC-MGM-2 on 1000 node graph-coloring problems. Networks of agents with average link densities (number of edges per agent) of 3 and 6 were generated, with randomly generated f- and g-constraints. In each case the g-budget assigned to all agents was varied between 0 to 50, as shown on the x-axis. Results were averaged over 30 runs of the algorithms. We first experiment with graphs where subgraphs of g-constraints had only two agents, i.e. the subgraphs $H$ of the DCOP constraint graph involved per g-constraint were of size two ($|H| = 2$). Figure 22 shows run-time in message cycles on a log-scale along the y-axis. For example, we see that for graphs of density 6, when g-budget is 10, the MC-MGM-2 algorithm “peaks” at about 500 cycles. The results demonstrate that MC-MGM-1 runs between 2.3 and 9.5 times faster than MC-MGM-2 on problems with a link density of 3, and 11.6 to 17.1 faster when the link density is 6. Global utility varied by less than 2% between the algorithms for both densities, as shown in Figure 23.
The second experiment focused on DCOP graphs with $|H| = 3$. The differences between MC-MGM-1 and 2 are more pronounced than with subgraphs of only 2 agents. We again found MC-MGM-1 is significantly faster than MC-MGM-2, as seen in Figure 24. For example, the difference in run-time peaks at a g-budget of 10, where MC-MGM-2 takes 132.7 times as long to terminate as MC-MGM-1. Figure 25 show the global utility illustrating that MC-MGM-2 resulted in solutions on average 49% and 6% higher in global utility for densities 6 and 3, respectively. At its peak, MC-MGM-2’s solution had roughly twice the global utility, when the link density was 6, indicating that MC-MGM-2 can indeed provide significantly higher utility in more complex graphs. A more thorough analysis of relationship of graph structure and the utilities obtained via mc-mgm-k algorithms remains an issue for future work.

The key conclusion from this scale-up investigation is this: compared to 1000s of cycles taken by complete algorithms attacking MC-DCOPs of 20-30 nodes, this is a 50-fold scale-up in number of nodes running with a 10-fold speedup. There is a loss in solution quality in incomplete algorithms, and thus
whether we choose to run complete or incomplete algorithms depends on the demands of the application.

6 Related Work

This section examines four main areas of work that is related to the research in this article: (i) efficiency work in Distributed Constraint Optimization (DCOP) algorithms; (ii) advances in multi-objective tradeoffs within DCOPs; and (iii) research on multi-criteria collaboration beyond DCOPs.

6.1 Efficiency in DCOPs

There is significant continued progress in singly-constrained complete and incomplete DCOP algorithms [31, 47, 50, 1, 38]. There are three leading complete algorithms, which all use slightly different mechanisms to solve the DCOP problem: Adopt, OptAPO and DPOP.

- **Adopt[31]**: This article already provided detailed background information on Adopt and the complete multiply-constrained DCOP algorithms developed as part of this article built upon the Adopt algorithm. Other work has continued to build upon and improve Adopt. For example, Maheswaran et al. [24] proposed node ordering heuristics, to construct shallower trees in Adopt, which allowed for greater efficiency. Ali et al. [1] presented preprocessing heuristics that propagate lower bounds on f-costs to variables in the Adopt DFS tree and allow for quicker convergence on the optimal solution. Additionally, Davin and Modi [7] provide techniques to efficiently handle multiple variables per agent in Adopt since previous work had tended to focus on problems with a single variable per agent. Yeoh et al. [45] presented a variant of Adopt called BnB-Adopt which does depth-first instead of best-first search and has been shown to significantly reduce the redundant computation in Adopt. Finally, Yeoh et al. [46] presented variants of Adopt and BnB-Adopt with various caching schemes that tradeoff limited amount of extra space for gains in efficiency.

- **NCBB[4]**: No-Commitment Branch and Bound (NCBB) is a newer branch and bound algorithm that, like Adopt, only uses polynomial space. It exploits the inherent parallelism of DCOP problems by allowing different sub-trees to explore potential solutions assuming different sets of values for their ancestors. The algorithm has been shown to outperform the original Adopt algorithm, but it has not been compared to the more efficient versions that have since been developed [4].

- **OptAPO[27]**: OptAPO is a competitor to Adopt that makes use of a partial centralization mechanism. Adopt uses very little centralization in its execution, the main exception being the propagation of f-costs to the
root node. In OptAPO, partial centralization allows an individual variable to manage a group of other nodes by deciding their assignments; in the extreme case, a single node may manage assignments for all other nodes, leading to full centralization. There has been much debate about how to compare the efficiency of Adopt and OptAPO [27, 6] with different performance metrics giving an edge to one algorithm over the other. Ultimately, the relative costs of computation and communication in the domain appear to govern the choice of an appropriate algorithm.

- **DPOP** [38]: DPOP, another of the complete DCOP algorithms, is a variable elimination algorithm, where individual variables communicate all their information to their neighbor in one-shot. This contrasts with the repeated rounds of value and cost message exchange in Adopt. The result is that DPOP leads to a significantly smaller number of messages exchanged among agents; unfortunately, the size of each message grows exponentially in the link density of a variable and the domain size. DPOP’s exponential memory requirements are problematic in some domains, but its low levels of communication make it an option for domains where the more communication intensive algorithms are infeasible. Recent research in DPOP has attempted to reduce the memory requirements of the algorithm [39].

The work in this article is complementary to these advances. The multiply-constrained DCOP formulation presents a new challenge for all of these algorithms. 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 and NCBB. Whether DPOP [38] could benefit from the techniques introduced here is a challenge for future work. Algorithms such as DPOP may face challenges when addressing multiply-constrained DCOPs because the variable elimination algorithms rely on there being a single best response to any combinations of a variable’s ancestors’ values. However, in MC-DCOP, there are multiple best responses to a particular combination of ancestors’ values, one for each way the sets of g-budgets could be split. This would lead to a massive increase in the amount of information a variable would need to send its parent and thus to an explosion in the space requirements of the parent. Since space requirements are the main limiting factor on how well DPOP can scale, this would exacerbate an existing problem.

In addition to the complete DCOP algorithms, there have been significant advances in incomplete algorithms [34, 12, 43, 51, 48]. We focus on two leading incomplete DCOP algorithms: MGM and DSA.

- **MGM** [34, 26, 14]: The MGM algorithms (MGM-1 and MGM-2) developed by Pearce et al were extensively described earlier since they provide the basis for the new MC-MGM algorithms described earlier. Recent research on the algorithms has established that the algorithms are k-optimal and that two types of guarantees can be made about the quality of the solution reached by these algorithms [34]. The first type of guarantee fixes an upper bound on the number of k-optima that can occur in a problem.
The second type of guarantee establishes a lower bound on the quality of the k-optimum as a percentage of the quality the globally optimal solution. These results allow users to estimate the benefits of using a particular level of k-optimum on a particular problem because the efficiency/performance tradeoff has been rigorously quantified. Additionally, more recent work has focused on developing asynchronous versions of MGM-like algorithms for arbitrary $k$[22].

- **DSA[34, 26, 14]**: The Distributed Stochastic Algorithm (DSA) is similar to the MGM algorithms but introduces a stochastic element to the algorithm [14, 34]. Instead of an agent simply changing values when it has the highest local gain, an agent decides randomly whether to move or not. This reduces the messaging load of the algorithm and in some cases allows the agents to reach a higher quality k-optimum than MGM, while in others, MGM outperforms DSA.

Our work on MC-MGM is complementary to these advances and benefits from them. Furthermore, given that DSA and MGM are very similar, the techniques used to build MC-MGM would transfer fairly easily to DSA. The more recently developed Max-Sum algorithms[12] are promising to develop into another key family of incomplete DCOP algorithms; MC-MGM type techniques could potentially be of benefit to this family of algorithms as well.

### 6.2 Multiple Objectives within DCOPs

While this article focuses on multiple cost objectives (i.e. f-costs and g-costs), complete DCOP algorithms already engage in a complex tradeoff between privacy, efficiency, diversity and optimality. Maheswaran et al [25] and Greenstadt et al [17, 16] studied the privacy loss in different complete DCOP algorithms, thus providing a better understanding of the privacy-efficiency tradeoffs in these algorithms. Traditionally, DCOP algorithms tolerate some loss in privacy for the sake of efficiency. However, Silaghi [40] coined the term “semi-cooperative algorithms” to describe DCOP algorithms that required agents to attain global optimality while maintaining privacy. Yokoo et al.[49] looked at how to use multiple external server agents to find the global optimal while guaranteeing significant privacy. However, this guarantee came at the cost of efficiency.

The balancing of multiple objectives within regular DCOP algorithms also comes to light in the work on incomplete algorithms. Incomplete algorithms[34, 43, 51, 48] trade off optimality to gain efficiency. Pearce et al.’s k-optimality [34] presented a novel consideration of multiple objectives within DCOPs: solution quality and solution diversity. Thus, multiple solutions are chosen not only due to their low cost but also their diversity, i.e. how far away the solutions are from each other. Thus, Pearce et al’s technique focuses on choosing multiple DCOP solutions that are each locally optimal, and are guaranteed to be each a distance of k agent’s value assignments away from other solutions. Such tradeoffs in optimality for efficiency are also seen in the error-bounds in Adopt [31], which allow it to trade the optimality of a solution for an improvement in efficiency.
There is, therefore, a vast space of tradeoffs in DCOP algorithms, and the MC-DCOP algorithms in this article explore one area within this space. Indeed, the algorithms in this article add another dimension to this tradeoff space, that of resource constraints. Furthermore, this article has explored the privacy-efficiency tradeoff as it occurs in MC-DCOP domains by examining the effects of having fine-grained control over the privacy of resource constraints. This extends beyond the whole algorithm approaches that have been used in studying the privacy-efficiency tradeoff in regular DCOP algorithms [16, 25, 17].

This article briefly touched on another area where two objectives require balancing: semi-cooperativeness. Semi-cooperativeness involves trading off team objectives for self-interest. In general, multi-agent coordination work has fallen into two categories, those where agents are assumed to be fully cooperative, like in DCOP [31, 1, 38, 27], or these where agents are fully self-interested, as in game theory [8]. Recently, some distributed constraint reasoning or related algorithms have been built that do not assume either fully cooperative or fully self-interested agents [11, 52, 13]; our article complements this work by focusing on the algorithmic efficiency of complete and incomplete DCOP algorithms when faced with bounded optimization.

6.3 Multi-criteria collaboration in general

Also related to the multiply-constrained DCOP work in this article is research into multi-criteria collaboration [32, 10, 33, 19, 28] which looks at finding a pareto-optimal for problems containing multiple objectives rather than optimizing a single objective subject to resource constraints. While that work has focused on applications such as distributed planning, it did not benefit from recent research that formalized DCOP and developed efficient algorithms for it. The algorithms in this article built on these efficient DCOP algorithms. Approaches to multi-criteria constraint satisfaction and optimization problems have tackled the problem using centralized methods [15, 20], but a central contribution of this article is in tackling the distributed problem, which requires designing algorithms where agents function without global knowledge [20].

6.4 More Recent Work

Several papers have built on this work since the early version of it was published as a conference paper [3].

- Pecora et al proposed an algorithm that is very similar to MCAP but which decreased the expansion of the DFS tree by merging the virtual variables with the lowest priority node in a resource constraint. The algorithm extends MCAP by allowing the dynamic posting of resource constraints. However, it cannot maintain the privacy of constraints or perform pre-emptive search space pruning as in MCAS [36, 37].

- Matsui et al proposed an alternate implementation of MCAS that decreased the restructuring of the DFS tree by merging the virtual variables
with a node’s parent instead of placing them lower in the priority tree. The algorithm runs more quickly than MCAS, but it only considers unary resource constraints rather than the n-ary ones modeled here and privacy is not considered [29]. To illustrate this difference more concretely, consider the following simple example. In Matsui et al’s work, given a pair of variables, $x_1$ and $x_2$, that are neighbors, the resources consumed (the g-cost) by an agent owning $x_1$ are only dependent on the value assigned to variable $x_1$. In contrast, our work models a more general case, where the resources consumed may be based on the values assigned to both $x_1$ and $x_2$ (or in general n-ary combinations of values).

7 Conclusion

This article focused on extending the expressivity of DCOP to capture more complex domains, in particular distributed bounded optimization domains. These domains require that networks of agents not only optimize a global objective function as in DCOP, but also satisfy local resource constraints. Existing DCOP algorithms are unable to solve bounded optimization problems because they are unable to handle local, possibly private resource constraints in addition to the team goal. This article defined the MC-DCOP framework, which is an extension to DCOP, to capture these problems and described three new algorithms for solving the MC-DCOP problems.

The goal in designing the algorithms was to prevent a significant increase in runtime complexity by using the information inherent in the resource constraints to preemptively prune the search space for the global goal. In order to obtain the maximum benefit, it was useful to have as many agents as possible make their resource information available. Thus, giving fine-grained control over the privacy of resource constraints was important. The other tradeoff to be considered in designing the new algorithms was scalability vs. optimality. Some bounded optimization problems require complete algorithms which can find the most effective use of scarce resources, while others require incomplete algorithms which can rapidly solve large-scale problems at the cost of finding suboptimal solutions.

Three different algorithms were developed: one complete algorithm and two incomplete algorithms. The complete algorithm, Multiply-Constrained Adopt (MCA), found the globally-optimal solution to bounded optimization problems. The two incomplete algorithms differed in the locality of the optimal that they were designed to reach. Multiply-Constrained Maximum Gain Message-1 (MC-MGM-1) was a 1-optimal algorithm which only considered moves that were profitable for an individual agent. MC-MGM-2 allowed pairs of agents to make coordinated moves, which allowed it to break out of some of the lower quality local optima that MC-MGM-1 could not escape. All three algorithms allowed for fine-grained control over the privacy vs. efficiency tradeoff. There were four main algorithmic innovations in designing the multiply-constrained DCOP algorithms: (i) transforming the network to allow private n-ary constraints to
be enforced; (ii) assigning upper-bounds on resource consumption to neighbors, in order to gain efficiency; (iii) identifying a structural property of the graph – T-nodes – which allowed agents to calculate exact bounds on resource consumption; (iv) using a virtual value assignment to identify when resource constraints have rendered a problem unsatisfiable. Proofs of correctness were provided for both sets of algorithms. Experiments were run to demonstrate the efficacy of the MC-DCOP algorithms at solving bounded optimization problems as well as to explore the tradeoffs that were built into the algorithms: privacy vs. efficiency and optimality vs. scalability. Experimental results confirmed the usefulness of fine-grained control over the privacy/efficiency tradeoff. Based on the results presented in this article, developing more efficient complete algorithms for multiply-constrained DCOPs, and developing incomplete algorithms for higher “k” (e.g. MC-MGM-3 or MC-MGM-4 or higher) remain issues for future work.

8 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


