A Complete Distributed Constraint Optimization Method For Non-Traditional Pseudotree Arrangements*

James Atlas
Computer and Information Sciences
University of Delaware
Newark, DE 19716
atlas@cis.udel.edu

Keith Decker
Computer and Information Sciences
University of Delaware
Newark, DE 19716
decker@cis.udel.edu

ABSTRACT

Distributed Constraint Optimization (DCOP) is a general framework that can model complex problems in multi-agent systems. Several current algorithms that solve general DCOP instances, including ADOPT and DPOP, arrange agents into a traditional pseudotree structure. We introduce an extension to the DPOP algorithm that handles an extended set of pseudotree arrangements. Our algorithm correctly solves DCOP instances for pseudotrees that include edges between nodes in separate branches. The algorithm also solves instances with traditional pseudotree arrangements using the same procedure as DPOP.

We compare our algorithm with DPOP using several metrics including the induced width of the pseudotrees, the maximum dimensionality of messages and computation, and the maximum sequential path cost through the algorithm. We prove that for some problem instances it is not possible to generate a traditional pseudotree using edge-traversal heuristics that will outperform a cross-edged pseudotree. We use multiple heuristics to generate pseudotrees and choose the best pseudotree in linear space-time complexity. For some problem instances we observe significant improvements in message and computation sizes compared to DPOP.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent Systems*

General Terms

Algorithms

*This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under the COORDINATORs program and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-05-C-0034, subcontract 01SC-FA8750-05-C0034. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or AFRL.

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

Keywords

Distributed Constraint Satisfaction and Optimization, Multi-agent Coordination

1. INTRODUCTION

Many historical problems in the AI community can be transformed into Constraint Satisfaction Problems (CSP). With the advent of distributed AI, multi-agent systems became a popular way to model the complex interactions and coordination required to solve distributed problems. CSPs were originally extended to distributed agent environments in [9]. Early domains for distributed constraint satisfaction problems (DisCSP) included job shop scheduling [1] and resource allocation [2]. Many domains for agent systems, especially teamwork coordination, distributed scheduling, and sensor networks, involve overly constrained problems that are difficult or impossible to satisfy for every constraint.

Recent approaches to solving problems in these domains rely on optimization techniques that map constraints into multi-valued utility functions. Instead of finding an assignment that satisfies all constraints, these approaches find an assignment that produces a high level of global utility. This extension to the original DisCSP approach has become popular in multi-agent systems, and has been labeled the Distributed Constraint Optimization Problem (DCOP) [1].

Current algorithms that solve complete DCOPs use two main approaches: search and dynamic programming. Search based algorithms that originated from DisCSP typically use some form of backtracking [10] or bounds propagation, as in ADOPT [3]. Dynamic programming based algorithms include DPOP and its extensions [5, 6, 7]. To date, both categories of algorithms arrange agents into a traditional pseudotree to solve the problem.

It has been shown in [6] that any constraint graph can be mapped into a traditional pseudotree. However, it was also shown that finding the optimal pseudotree was NP-Hard. We began to investigate the performance of traditional pseudotrees generated by current edge-traversal heuristics. We found that these heuristics often produced little parallelism as the pseudotrees tended to have high depth and low branching factors. We suspected that there could be other ways to arrange the pseudotrees that would provide increased parallelism and smaller message sizes. After exploring these other arrangements we found that cross-edged pseudotrees provide shorter depths and higher branching factors than the traditional pseudotrees. Our hypothesis was that these cross-edged pseudotrees would outperform traditional pseudotrees for some problem types.

In this paper we introduce an extension to the DPOP algorithm that handles an extended set of pseudotree arrangements which include cross-edged pseudotrees. We begin with a definition of

978-81-904262-7-5 (RPS)

DCOP, traditional pseudotrees, and cross-edged pseudotrees. We then provide a summary of the original DPOP algorithm and introduce our DCPOP algorithm. We discuss the complexity of our algorithm as well as the impact of pseudotree generation heuristics. We then show that our Distributed Cross-edged Pseudotree Optimization Procedure (DCPOP) performs significantly better in practice than the original DPOP algorithm for some problem instances. We conclude with a selection of ideas for future work and extensions for DCPOP.

2. PROBLEM DEFINITION

DCOP has been formalized in slightly different ways in recent literature, so we will adopt the definition as presented in [6]. A Distributed Constraint Optimization Problem with n nodes and m constraints consists of the tuple $< X, D, U >$ where:

- X = $\{x_1,..,x_n\}$ is a set of variables, each one assigned to a unique agent
- D = $\{d_1,..,d_n\}$ is a set of finite domains for each variable
- U = $\{u_1,..,u_m\}$ is a set of utility functions such that each function involves a subset of variables in X and defines a utility for each combination of values among these variables

An optimal solution to a DCOP instance consists of an assignment of values in D to X such that the sum of utilities in U is maximal. Problem domains that require minimum cost instead of maximum utility can map costs into negative utilities. The utility functions represent soft constraints but can also represent hard constraints by using arbitrarily large negative values. For this paper we only consider binary utility functions involving two variables. Higher order utility functions can be modeled with minor changes to the algorithm, but they also substantially increase the complexity.

2.1 Traditional Pseudotrees

Pseudotrees are a common structure used in search procedures to allow parallel processing of independent branches. As defined in [6], a pseudotree is an arrangement of a graph G into a rooted tree T such that vertices in G that share an edge are in the same branch in T. A back-edge is an edge between a node X and any node which lies on the path from X to the root (excluding X's parent). Figure 1 shows a pseudotree with four nodes, three edges (A-B, B-C, B-D), and one back-edge (A-C). Also defined in [6] are four types of relationships between nodes exist in a pseudotree:

- P(X) - the parent of a node X: the single node higher in the pseudotree that is connected to X directly through a tree edge
- C(X) - the children of a node X: the set of nodes lower in the pseudotree that are connected to X directly through tree edges
- PP(X) - the pseudo-parents of a node X: the set of nodes higher in the pseudotree that are connected to X directly through back-edges (In Figure 1, A = PP(C))
- PC(X) - the pseudo-children of a node X: the set of nodes lower in the pseudotree that are connected to X directly through back-edges (In Figure 1, C = PC(A))

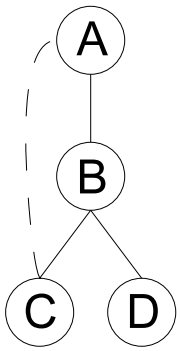

Figure 1: A traditional pseudotree. Solid line edges represent parent-child relationships and the dashed line represents a pseudo-parent-pseudo-child relationship.

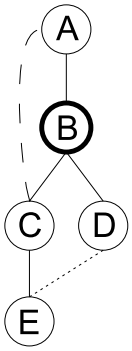

Figure 2: A cross-edged pseudotree. Solid line edges represent parent-child relationships, the dashed line represents a pseudo-parent-pseudo-child relationship, and the dotted line represents a branch-parent-branch-child relationship. The bolded node, B, is the merge point for node E.

2.2 Cross-edged Pseudotrees

We define a cross-edge as an edge from node X to a node Y that is above X but not in the path from X to the root. A cross-edged pseudotree is a traditional pseudotree with the addition of cross-edges. Figure 2 shows a cross-edged pseudotree with a cross-edge (D-E). In a cross-edged pseudotree we designate certain edges as primary. The set of primary edges defines a spanning tree of the nodes. The parent, child, pseudo-parent, and pseudo-child relationships from the traditional pseudotree are now defined in the context of this primary edge spanning tree. This definition also yields two additional types of relationships that may exist between nodes:

- BP(X) - the branch-parents of a node X: the set of nodes higher in the pseudotree that are connected to X but are not in the primary path from X to the root (In Figure 2, D = BP(E))
- BC(X) - the branch-children of a node X: the set of nodes lower in the pseudotree that are connected to X but are not in any primary path from X to any leaf node (In Figure 2, E = BC(D))

2.3 Pseudotree Generation

Current algorithms usually have a pre-execution phase to generate a traditional pseudotree from a general DCOP instance. Our DCPOP algorithm generates a cross-edged pseudotree in the same fashion. First, the DCOP instance $< X, D, U >$ translates directly into a graph with X as the set of vertices and an edge for each pair of variables represented in U. Next, various heuristics are used to arrange this graph into a pseudotree. One common heuristic is to perform a guided depth-first search (DFS) as the resulting traversal is a pseudotree, and a DFS can easily be performed in a distributed fashion. We define an edge-traversal based method as any method that produces a pseudotree in which all parent/child pairs share an edge in the original graph. This includes DFS, breadth-first search, and best-first search based traversals. Our heuristics that generate cross-edged pseudotrees use a distributed best-first search traversal.

3. DPOP ALGORITHM

The original DPOP algorithm operates in three main phases. The first phase generates a traditional pseudotree from the DCOP instance using a distributed algorithm. The second phase joins utility hypercubes from children and the local node and propagates them towards the root. The third phase chooses an assignment for each domain in a top down fashion beginning with the agent at the root node.

The complexity of DPOP depends on the size of the largest computation and utility message during phase two. It has been shown that this size directly corresponds to the induced width of the pseudotree generated in phase one [6]. DPOP uses polynomial time heuristics to generate the pseudotree since finding the minimum induced width pseudotree is NP-hard. Several distributed edge-traversal heuristics have been developed to find low width pseudotrees [8]. At the end of the first phase, each agent knows its parent, children, pseudo-parents, and pseudo-children.

3.1 Utility Propagation

Agents located at leaf nodes in the pseudotree begin the process by calculating a local utility hypercube. This hypercube at node X contains summed utilities for each combination of values in the domains for P(X) and PP(X). This hypercube has dimensional size equal to the number of pseudo-parents plus one. A message containing this hypercube is sent to P(X). Agents located at non-leaf nodes wait for all messages from children to arrive. Once the agent at node Y has all utility messages, it calculates its local utility hypercube which includes domains for P(Y), PP(Y), and Y. The local utility hypercube is then joined with all of the hypercubes from the child messages. At this point all utilities involving node Y are known, and the domain for Y may be safely eliminated from the joined hypercube. This elimination process chooses the best utility over the domain of Y for each combination of the remaining domains. A message containing this hypercube is now sent to P(Y). The dimensional size of this hypercube depends on the number of overlapping domains in received messages and the local utility hypercube. This dynamic programming based propagation phase continues until the agent at the root node of the pseudotree has received all messages from its children.

3.2 Value Propagation

Value propagation begins when the agent at the root node Z has received all messages from its children. Since Z has no parents or pseudo-parents, it simply combines the utility hypercubes received from its children. The combined hypercube contains only values for the domain for Z. At this point the agent at node Z simply chooses the assignment for its domain that has the best utility. A value propagation message with this assignment is sent to each node in C(Z). Each other node then receives a value propagation message from its parent and chooses the assignment for its domain that has the best utility given the assignments received in the message. The node adds its domain assignment to the assignments it received and passes the set of assignments to its children. The algorithm is complete when all nodes have chosen an assignment for their domain.

4. DCPOP ALGORITHM

Our extension to the original DPOP algorithm, shown in Algorithm 1, shares the same three phases. The first phase generates the cross-edged pseudotree for the DCOP instance. The second phase merges branches and propagates the utility hypercubes. The third phase chooses assignments for domains at branch merge points and in a top down fashion, beginning with the agent at the root node.

For the first phase we generate a pseudotree using several distributed heuristics and select the one with lowest overall complexity. The complexity of the computation and utility message size in DCPOP does not directly correspond to the induced width of the cross-edged pseudotree. Instead, we use a polynomial time method for calculating the maximum computation and utility message size for a given cross-edged pseudotree. A description of this method and the pseudotree selection process appears in Section 5. At the end of the first phase, each agent knows its parent, children, pseudo-parents, pseudo-children, branch-parents, and branch-children.

4.1 Merging Branches and Utility Propagation

In the original DPOP algorithm a node X only had utility functions involving its parent and its pseudo-parents. In DCPOP, a node X is allowed to have a utility function involving a branch-parent. The concept of a branch can be seen in Figure 2 with node E representing our node X. The two distinct paths from node E to node B are called branches of E. The single node where all branches of E meet is node B, which is called the merge point of E.

Agents with nodes that have branch-parents begin by sending a utility propagation message to each branch-parent. This message includes a two dimensional utility hypercube with domains for the node X and the branch-parent BP(X). It also includes a branch information structure which contains the origination node of the branch, X, the total number of branches originating from X, and the number of branches originating from X that are merged into a single representation by this branch information structure (this number starts at 1). Intuitively when the number of merged branches equals the total number of originating branches, the algorithm has reached the merge point for X. In Figure 2, node E sends a utility propagation message to its branch-parent, node D. This message has dimensions for the domains of E and D, and includes branch information with an origin of E, 2 total branches, and 1 merged branch.

As in the original DPOP utility propagation phase, an agent at leaf node X sends a utility propagation message to its parent. In DCPOP this message contains dimensions for the domains of P(X) and PP(X). If node X also has branch-parents, then the utility propagation message also contains a dimension for the domain of X, and will include a branch information structure. In Figure 2, node E sends a utility propagation message to its parent, node C. This message has dimensions for the domains of E and C, and includes branch information with an origin of E, 2 total branches, and 1 merged branch.

When a node Y receives utility propagation messages from all of

its children and branch-children, it merges any branches with the same origination node X. The merged branch information structure accumulates the number of merged branches for X. If the cumulative total number of merged branches equals the total number of branches, then Y is the merge point for X. This means that the utility hypercubes present at Y contain all information about the valuations for utility functions involving node X. In addition to the typical elimination of the domain of Y from the utility hypercubes, we can now safely eliminate the domain of X from the utility hypercubes. To illustrate this process, we will examine what happens in the second phase for node B in Figure 2.

In the second phase Node B receives two utility propagation messages. The first comes from node C and includes dimensions for domains E, B, and A. It also has a branch information structure with origin of E, 2 total branches, and 1 merged branch. The second comes from node D and includes dimensions for domains E and B. It also has a branch information structure with origin of E, 2 total branches, and 1 merged branch. Node B then merges the branch information structures from both messages because they have the same origination, node E. Since the number of merged branches originating from E is now 2 and the total branches originating from E is 2, node B now eliminates the dimensions for domain E. Node B also eliminates the dimension for its own domain, leaving only information about domain A. Node B then sends a utility propagation message to node A, containing only one dimension for the domain of A.

Although not possible in DPOP, this method of utility propagation and dimension elimination may produce hypercubes at node Y that do not share any domains. In DCPOP we do not join domain independent hypercubes, but instead may send multiple hypercubes in the utility propagation message sent to the parent of Y. This lazy approach to joins helps to reduce message sizes.

4.2 Value Propagation

As in DPOP, value propagation begins when the agent at the root node Z has received all messages from its children. At this point the agent at node Z chooses the assignment for its domain that has the best utility. If Z is the merge point for the branches of some node X, Z will also choose the assignment for the domain of X. Thus any node that is a merge point will choose assignments for a domain other than its own. These assignments are then passed down the primary edge hierarchy. If node X in the hierarchy has branch-parents, then the value assignment message from P(X) will contain an assignment for the domain of X. Every node in the hierarchy adds any assignments it has chosen to the ones it received and passes the set of assignments to its children. The algorithm is complete when all nodes have chosen or received an assignment for their domain.

4.3 Proof of Correctness

We will prove the correctness of DCPOP by first noting that DCPOP fully extends DPOP and then examining the two cases for value assignment in DCPOP. Given a traditional pseudotree as input, the DCPOP algorithm execution is identical to DPOP. Using a traditional pseudotree arrangement no nodes have branch-parents or branch-children since all edges are either back-edges or tree edges. Thus the DCPOP algorithm using a traditional pseudotree sends only utility propagation messages that contain domains belonging to the parent or pseudo-parents of a node. Since no node has any branch-parents, no branches exist, and thus no node serves as a merge point for any other node. Thus all value propagation assignments are chosen at the node of the assignment domain.

For DCPOP execution with cross-edged pseudotrees, some nodes serve as merge points. We note that any node X that is not a merge point assigns its value exactly as in DPOP. The local utility hypercube at X contains domains for X, P(X), PP(X), and BC(X). As in DPOP the value assignment message received at X includes the values assigned to P(X) and PP(X). Also, since X is not a merge point, all assignments to BC(X) must have been calculated at merge points higher in the tree and are in the value assignment message from P(X). Thus after eliminating domains for which assignments are known, only the domain of X is left. The agent at node X can now correctly choose the assignment with maximum utility for its own domain.

If node X is a merge point for some branch-child Y, we know that X must be a node along the path from Y to the root, and from P(Y) and all BP(Y) to the root. From the algorithm, we know that Y necessarily has all information from C(Y), PC(Y), and BC(Y) since it waits for their messages. Node X has information about all nodes below it in the tree, which would include Y, P(Y), BP(Y), and those PP(Y) that are below X in the tree. For any PP(Y) above X in the tree, X receives the assignment for the domain of PP(Y) in the value assignment message from P(X). Thus X has utility information about all of the utility functions of which Y is a part. By eliminating domains included in the value assignment message, node X is left with a local utility hypercube with domains for X and Y. The agent at node X can now correctly choose the assignments with maximum utility for the domains of X and Y.

4.4 Complexity Analysis

The first phase of DCPOP sends one message to each P(X), PP(X), and BP(X). The second phase sends one value assignment message to each C(X). Thus, DCPOP produces a linear number of messages with respect to the number of edges (utility functions) in the cross-edged pseudotree and the original DCOP instance. The actual complexity of DCPOP depends on two additional measurements: message size and computation size.

Message size and computation size in DCPOP depend on the number of overlapping branches as well as the number of overlapping back-edges. It was shown in [6] that the number of overlapping back-edges is equal to the induced width of the pseudotree. In a poorly constructed cross-edged pseudotree, the number of overlapping branches at node X can be as large as the total number of descendants of X. Thus, the total message size in DCPOP in a poorly constructed instance can be space-exponential in the total number of nodes in the graph. However, in practice a well constructed cross-edged pseudotree can achieve much better results. Later we address the issue of choosing well constructed cross-edged pseudotrees from a set.

We introduce an additional measurement of the maximum sequential path cost through the algorithm. This measurement directly relates to the maximum amount of parallelism achievable by the algorithm. To take this measurement we first store the total computation size for each node during phase two and three. This computation size represents the number of individual accesses to a value in a hypercube at each node. For example, a join between two domains of size 4 costs $4 * 4 = 16$. Two directed acyclic graphs (DAG) can then be drawn; one with the utility propagation messages as edges and the phase two costs at nodes, and the other with value assignment messages and the phase three costs at nodes. The maximum sequential path cost is equal to the sum of the longest path on each DAG from the root to any leaf node.

5. HEURISTICS

In our assessment of complexity in DCPOP we focused on the worst case possibly produced by the algorithm. We acknowledge

Algorithm 1 DCPOP Algorithm

1: **DCPOP**($X; D; U$)
Each agent X_i executes:

Phase 1: pseudotree creation
2: elect leader from all $X_j \in X$
3: elected leader initiates pseudotree creation
4: afterwards, X_i knows P(X_i), PP(X_i), BP(X_i), C(X_i), BC(X_i) and PC(X_i)

Phase 2: UTIL message propagation
5: **if** $|\text{BP}(X_i)| > 0$ **then**
6: BRANCH$_{X_i} \leftarrow |\text{BP}(X_i)| + 1$
7: **for all** $X_k \in$BP(X_i) **do**
8: UTIL$_{X_i}(X_k) \leftarrow$Compute_utils(X_i, X_k)
9: Send_message(X_k,UTIL$_{X_i}(X_k)$,BRANCH$_{X_i}$)
10: **if** $|\text{C}(X_i)| = 0$(i.e. X_i is a leaf node) **then**
11: UTIL$_{X_i}$(P(X_i)) $\leftarrow$ Compute_utils(P(X_i),PP(X_i)) for all PP(X_i)
12: Send_message(P(X_i), UTIL$_{X_i}$(P(X_i)),BRANCH$_{X_i}$)
13: Send_message(PP(X_i), $empty\_UTIL$, $empty\_BRANCH$) to all PP(X_i)
14: activate UTIL_Message_handler()

Phase 3: VALUE message propagation
15: activate VALUE_Message_handler()
END ALGORITHM

UTIL_Message_handler(X_k,**UTIL**$_{X_k}(X_i)$, **BRANCH**$_{X_k}$)
16: store UTIL$_{X_k}(X_i)$,BRANCH$_{X_k}(X_i)$
17: **if** UTIL messages from all children and branch children arrived **then**
18: **for all** $B_j \in$BRANCH(X_i) **do**
19: **if** B_j is merged **then**
20: join all hypercubes where $B_j \in$UTIL(X_i)
21: eliminate B_j from the joined hypercube
22: **if** P(X_i) $== null$ (that means X_i is the root) **then**
23: $v * i \leftarrow$ Choose_optimal($null$)
24: Send VALUE($X_i, v * i$) to all C(X_i)
25: **else**
26: UTIL$_{X_i}$(P(X_i)) $\leftarrow$ Compute_utils(P(X_i), PP(X_i))
27: Send_message(P(X_i),UTIL$_{X_i}$(P(X_i)), BRANCH$_{X_i}$(P(X_i)))

VALUE_Message_handler(**VALUE**$_{X_i}$,**P**(X_i))
28: add all $X_k \leftarrow v * k \in$VALUE$_{X_i}$,P(X_i) to $agent\_view$
29: $X_i \leftarrow v * i =$Choose_optimal($agent\_view$)
30: Send VALUE$_{X_l}, X_i$ to all $X_l \in$C(X_i)

that in real world problems the generation of the pseudotree has a significant impact on the actual performance. The problem of finding the best pseudotree for a given DCOP instance is NP-Hard. Thus a heuristic is used for generation, and the performance of the algorithm depends on the pseudotree found by the heuristic. Some previous research focused on finding heuristics to generate good pseudotrees [8]. While we have developed some heuristics that generate good cross-edged pseudotrees for use with DCPOP, our focus has been to use multiple heuristics and then select the best pseudotree from the generated pseudotrees.

We consider only heuristics that run in polynomial time with respect to the number of nodes in the original DCOP instance. The actual DCPOP algorithm has worst case exponential complexity, but we can calculate the maximum message size, computation size, and sequential path cost for a given cross-edged pseudotree in linear space-time complexity. To do this, we simply run the algorithm without attempting to calculate any of the local utility hypercubes or optimal value assignments. Instead, messages include dimensional and branch information but no utility hypercubes.

After each heuristic completes its generation of a pseudotree, we execute the measurement procedure and propagate the measurement information up to the chosen root in that pseudotree. The root then broadcasts the total complexity for that heuristic to all nodes. After all heuristics have had a chance to complete, every node knows which heuristic produced the best pseudotree. Each node then proceeds to begin the DCPOP algorithm using its knowledge of the pseudotree generated by the best heuristic.

The heuristics used to generate traditional pseudotrees perform a distributed DFS traversal. The general distributed algorithm uses a token passing mechanism and a linear number of messages. Improved DFS based heuristics use a special procedure to choose the root node, and also provide an ordering function over the neighbors of a node to determine the order of path recursion. The DFS based heuristics used in our experiments come from the work done in [4, 8].

5.1 The best-first cross-edged pseudotree heuristic

The heuristics used to generate cross-edged pseudotrees perform a best-first traversal. A general distributed best-first algorithm for node expansion is presented in Algorithm 2. An evaluation function at each node provides the values that are used to determine the next best node to expand. Note that in this algorithm each node only exchanges its best value with its neighbors. In our experiments we used several evaluation functions that took as arguments an ordered list of ancestors and a node, which contains a list of neighbors (with each neighbor's placement depth in the tree if it was placed). From these we can calculate branch-parents, branch-children, and unknown relationships for a potential node placement. The best overall function calculated the value as $ancestors - (branchparents + branchchildren)$ with the number of unknown relationships being a tiebreak. After completion each node has knowledge of its parent and ancestors, so it can easily determine which connected nodes are pseudo-parents, branch-parents, pseudo-children, and branch-children.

The complexity of the best-first traversal depends on the complexity of the evaluation function. Assuming a complexity of O(V) for the evaluation function, which is the case for our best overall function, the best-first traversal is O($V \cdot E$) which is at worst O(n^3). For each $v \in V$ we perform a place operation, and find the next node to place using the getBestNeighbor operation. The place operation is at most O(V) because of the sent messages. Finding the next node uses recursion and traverses only already placed

Algorithm 2 Distributed Best-First Search Algorithm

$root \leftarrow electedleader$
$next(root, \emptyset)$

place(node, parent)
$node.parent \leftarrow parent$
$node.ancestors \leftarrow parent.ancestors \cup parent$
send placement message $(node, node.ancestors)$ to all neighbors of $node$

next(current, previous)
if current is not placed **then**
  $place(current, previous)$
  $next(current, \emptyset)$
else
  $best \leftarrow getBestNeighbor(current, previous)$
  if $best = \emptyset$ **then**
    if $previous = \emptyset$ **then**
      terminate, all nodes are placed
    $next(previous, \emptyset)$
  else
    $next(best, current)$

getBestNeighbor(current, previous)
$best \leftarrow \emptyset; score \leftarrow 0$
for all $n \in current.neighbors$ **do**
  if $n! = previous$ **then**
    if n is placed **then**
      $nscore \leftarrow getBestNeighbor(n, current)$
    else
      $nscore \leftarrow evaluate(current, n)$
    if $nscore > score$ **then**
      $score \leftarrow nscore$
      $best \leftarrow n$
return $best, score$

nodes, so it has O(V) recursions. Each recursion performs a recursive getBestNeighbor operation that traverses all placed nodes and their neighbors. This operation is O($V \cdot E$), but results can be cached using only O(V) space at each node. Thus we have O($V \cdot (V+V+V \cdot E)$) = O($V^2 \cdot E$). If we are smart about evaluating local changes when each node receives placement messages from its neighbors and cache the results the getBestNeighbor operation is only O(E). This increases the complexity of the place operation, but for all placements the total complexity is only O($V \cdot E$). Thus we have an overall complexity of O($V \cdot E + V \cdot (V+E)$) = O($V \cdot E$).

6. COMPARISON OF COMPLEXITY IN DPOP AND DCPOP

We have already shown that given the same input, DCPOP performs the same as DPOP. We also have shown that we can accurately predict performance of a given pseudotree in linear space-time complexity. If we use a constant number of heuristics to generate the set of pseudotrees, we can choose the best pseudotree in linear space-time complexity. We will now show that there exists a DCOP instance for which a cross-edged pseudotree outperforms all possible traditional pseudotrees (based on edge-traversal heuristics).

In Figure 3(a) we have a DCOP instance with six nodes. This is a bipartite graph with each partition fully connected to the other

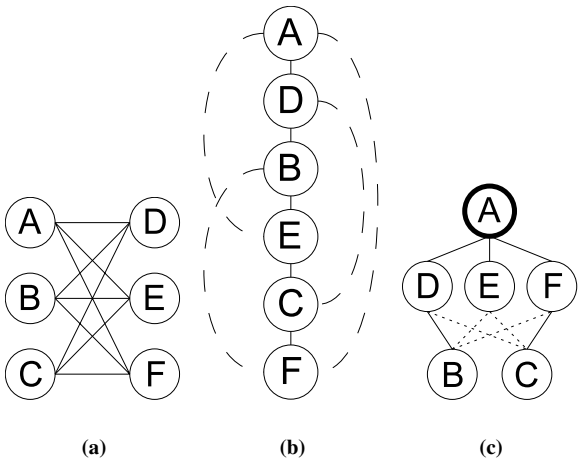

Figure 3: (a) The DCOP instance (b) A traditional pseudotree arrangement for the DCOP instance (c) A cross-edged pseudotree arrangement for the DCOP instance

partition. In Figure 3(b) we see a traditional pseudotree arrangement for this DCOP instance. It is easy to see that any edge-traversal based heuristic cannot expand two nodes from the same partition in succession. We also see that no node can have more than one child because any such arrangement would be an invalid pseudotree. Thus any traditional pseudotree arrangement for this DCOP instance must take the form of Figure 3(b). We can see that the back-edges F-B and F-A overlap node C. Node C also has a parent E, and a back-edge with D. Using the original DPOP algorithm (or DCPOP since they are identical in this case), we find that the computation at node C involves five domains: A, B, C, D, and E.

In contrast, the cross-edged pseudotree arrangement in Figure 3(c) requires only a maximum of four domains in any computation during DCPOP. Since node A is the merge point for branches from both B and C, we can see that each of the nodes D, E, and F have two overlapping branches. In addition each of these nodes has node A as its parent. Using the DCPOP algorithm we find that the computation at node D (or E or F) involves four domains: A, B, C, and D (or E or F).

Since no better traditional pseudotree arrangement can be created using an edge-traversal heuristic, we have shown that DCPOP can outperform DPOP even if we use the optimal pseudotree found through edge-traversal. We acknowledge that pseudotree arrangements that allow parent-child relationships without an actual constraint can solve the problem in Figure 3(a) with maximum computation size of four domains. However, current heuristics used with DPOP do not produce such pseudotrees, and such a heuristic would be difficult to distribute since each node would require information about nodes with which it has no constraint. Also, while we do not prove it here, cross-edged pseudotrees can produce smaller message sizes than such pseudotrees even if the computation size is similar. In practice, since finding the best pseudotree arrangement is NP-Hard, we find that heuristics that produce cross-edged pseudotrees often produce significantly smaller computation and message sizes.

7. EXPERIMENTAL RESULTS

Existing performance metrics for DCOP algorithms include the total number of messages, synchronous clock cycles, and message size. We have already shown that the total number of messages is linear with respect to the number of constraints in the DCOP instance. We also introduced the maximum sequential path cost (PC) as a measurement of the maximum amount of parallelism achievable by the algorithm. The maximum sequential path cost is equal to the sum of the computations performed on the longest path from the root to any leaf node. We also include as metrics the maximum computation size in number of dimensions (CD) and maximum message size in number of dimensions (MD). To analyze the relative complexity of a given DCOP instance, we find the minimum induced width (IW) of any traditional pseudotree produced by a heuristic for the original DPOP.

7.1 Generic DCOP instances

For our initial tests we randomly generated two sets of problems with 3000 cases in each. Each problem was generated by assigning a random number (picked from a range) of constraints to each variable. The generator then created binary constraints until each variable reached its maximum number of constraints. The first set uses 20 variables, and the best DPOP IW ranges from 1 to 16 with an average of 8.5. The second set uses 100 variables, and the best DPOP IW ranged from 2 to 68 with an average of 39.3. Since most of the problems in the second set were too complex to actually compute the solution, we took measurements of the metrics using the techniques described earlier in Section 5 without actually solving the problem. Results are shown for the first set in Table 1 and for the second set in Table 2.

For the two problem sets we split the cases into low density and high density categories. Low density cases consist of those problems that have a best DPOP IW less than or equal to half of the total number of nodes (e.g. $IW \leq 10$ for the 20 node problems and $IW \leq 50$ for the 100 node problems). High density problems consist of the remainder of the problem sets.

In both Table 1 and Table 2 we have listed performance metrics for the original DPOP algorithm, the DCPOP algorithm using only cross-edged pseudotrees (DCPOP-CE), and the DCPOP algorithm using traditional and cross-edged pseudotrees (DCPOP-All). The pseudotrees used for DPOP were generated using 5 heuristics: DFS, DFS_MCN, DFS_CLIQUE_MCN, DFS_MCN_DSTB, and DFS_MCN_BEC. These are all versions of the guided DFS traversal discussed in Section 5. The cross-edged pseudotrees used for DCPOP-CE were generated using 5 heuristics: MCN, LCN, MCN_A-B, LCN_A-B, and LCSG_A-B. These are all versions of the best-first traversal discussed in Section 5.

For both DPOP and DCPOP-CE we chose the best pseudotree produced by their respective 5 heuristics for each problem in the set. For DCPOP-All we chose the best pseudotree produced by all 10 heuristics for each problem in the set. For the CD and MD metrics the value shown is the average number of dimensions. For the PC metric the value shown is the natural logarithm of the maximum sequential path cost (since the actual value grows exponentially with the complexity of the problem).

The final row in both tables is a measurement of improvement of DCPOP-All over DPOP. For the CD and MD metrics the value shown is a reduction in number of dimensions. For the PC metric the value shown is a percentage reduction in the maximum sequential path cost ($\% = \frac{DPOP-DCPOP}{DCPOP} * 100$). Notice that DCPOP-All outperforms DPOP on all metrics. This logically follows from our earlier assertion that given the same input, DCPOP performs exactly the same as DPOP. Thus given the choice between the pseudotrees produced by all 10 heuristics, DCPOP-All will always out-

	Low Density			High Density		
Algorithm	CD	MD	PC	CD	MD	PC
DPOP	7.81	6.81	3.78	13.34	12.34	5.34
DCPOP-CE	7.94	6.73	3.74	12.83	11.43	5.07
DCPOP-All	7.62	6.49	3.66	12.72	11.36	5.05
Improvement	0.18	0.32	13%	0.62	0.98	36%

Table 1: 20 node problems

	Low Density			High Density		
Algorithm	CD	MD	PC	CD	MD	PC
DPOP	33.35	32.35	14.55	58.51	57.50	19.90
DCPOP-CE	33.49	29.17	15.22	57.11	50.03	20.01
DCPOP-All	32.35	29.57	14.10	56.33	51.17	18.84
Improvement	1.00	2.78	104%	2.18	6.33	256%

Table 2: 100 node problems

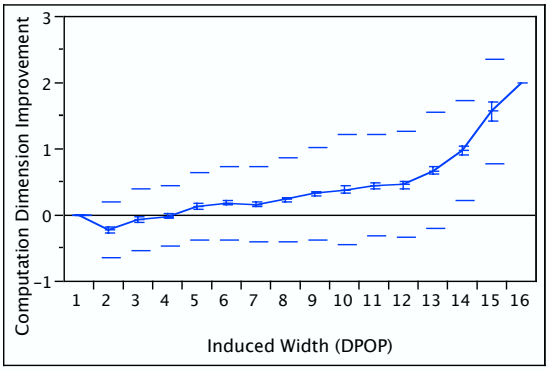

Figure 4: Computation Dimension Size

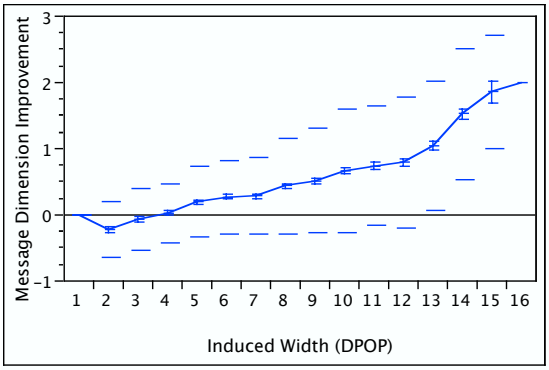

Figure 5: Message Dimension Size

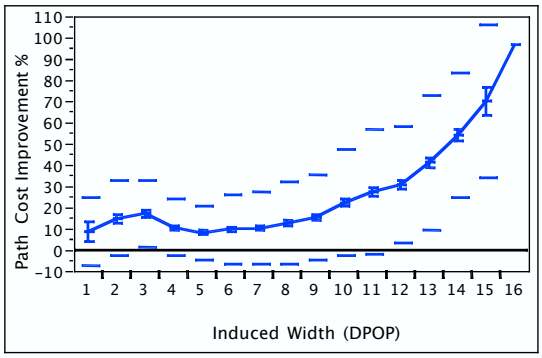

Figure 6: Path Cost

					DCPOP Improvement		
Ag	Mtg	Vars	Const	IW	CD	MD	PC
10	4	12	13.5	2.25	-0.01	-0.01	5.6%
30	14	44	57.6	3.63	0.09	0.09	10.9%
50	24	76	101.3	4.17	0.08	0.09	10.7%
100	49	156	212.9	5.04	0.16	0.20	30.0%
150	74	236	321.8	5.32	0.21	0.23	35.8%
200	99	316	434.2	5.66	0.18	0.22	29.5%

Table 3: Meeting Scheduling Problems

perform DPOP. Another trend we notice is that the improvement is greater for high density problems than low density problems. We show this trend in greater detail in Figures 4, 5, and 6. Notice how the improvement increases as the complexity of the problem increases.

7.2 Meeting Scheduling Problem

In addition to our initial generic DCOP tests, we ran a series of tests on the Meeting Scheduling Problem (MSP) as described in [6]. The problem setup includes a number of people that are grouped into departments. Each person must attend a specified number of meetings. Meetings can be held within departments or among departments, and can be assigned to one of eight time slots. The MSP maps to a DCOP instance where each variable represents the time slot that a specific person will attend a specific meeting. All variables that belong to the same person have mutual exclusion constraints placed so that the person cannot attend more than one meeting during the same time slot. All variables that belong to the same meeting have equality constraints so that all of the participants choose the same time slot. Unary constraints are placed on each variable to account for a person's valuation of each meeting and time slot.

For our tests we generated 100 sample problems for each combination of agents and meetings. Results are shown in Table 3. The values in the first five columns represent (in left to right order), the total number of agents, the total number of meetings, the total number of variables, the average total number of constraints, and the average minimum IW produced by a traditional pseudotree. The last three columns show the same metrics we used for the generic DCOP instances, except this time we only show the improvements of DCPOP-All over DPOP. Performance is better on average for all MSP instances, but again we see larger improvements for more complex problem instances.

8. CONCLUSIONS AND FUTURE WORK

We presented a complete, distributed algorithm that solves general DCOP instances using cross-edged pseudotree arrangements. Our algorithm extends the DPOP algorithm by adding additional utility propagation messages, and introducing the concept of branch merging during the utility propagation phase. Our algorithm also allows value assignments to occur at higher level merge points for lower level nodes. We have shown that DCPOP fully extends DPOP by performing the same operations given the same input. We have also shown through some examples and experimental data that DCPOP can achieve greater performance for some problem instances by extending the allowable input set to include cross-edged pseudotrees.

We placed particular emphasis on the role that edge-traversal heuristics play in the generation of pseudotrees. We have shown that the performance penalty is minimal to generate multiple heuristics, and that we can choose the best generated pseudotree in linear space-time complexity. Given the importance of a good pseudotree for performance, future work will include new heuristics to find better pseudotrees. Future work will also include adapting existing DPOP extensions [5, 7] that support different problem domains for use with DCPOP.

9. REFERENCES

[1] J. Liu and K. P. Sycara. Exploiting problem structure for distributed constraint optimization. In V. Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 246–254, San Francisco, CA, 1995. MIT Press.

[2] P. J. Modi, H. Jung, M. Tambe, W.-M. Shen, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. *Lecture Notes in Computer Science*, 2239:685–700, 2001.

[3] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS 03*, 2003.

[4] A. Petcu. Frodo: A framework for open/distributed constraint optimization. Technical Report No. 2006/001 2006/001, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), 2006. http://liawww.epfl.ch/frodo/.

[5] A. Petcu and B. Faltings. A-dpop: Approximations in distributed optimization. In *poster in CP 2005*, pages 802–806, Sitges, Spain, October 2005.

[6] A. Petcu and B. Faltings. Dpop: A scalable method for multiagent constraint optimization. In *IJCAI 05*, pages 266–271, Edinburgh, Scotland, Aug 2005.

[7] A. Petcu, B. Faltings, and D. Parkes. M-dpop: Faithful distributed implementation of efficient social choice problems. In *AAMAS 06*, pages 1397–1404, Hakodate, Japan, May 2006.

[8] G. Ushakov. Solving meeting scheduling problems using distributed pseudotree-optimization procedure. Master's thesis, École Polytechnique Fédérale de Lausanne, 2005.

[9] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[10] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.