Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

5-2011

Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems

William YEOH
New Mexico State University

Pradeep VARAKANTHAM
Singapore Management University, pradeepv@smu.edu.sg

Xiaoxun SUN
Google Inc

Sven KOENIG
University of Southern California

Follow this and additional works at: http://ink.library.smu.edu.sg/sis_research


Part of the Artificial Intelligence and Robotics Commons, Business Commons, and the Operations Research, Systems Engineering and Industrial Engineering Commons

Citation

YEOH, William; VARAKANTHAM, Pradeep; SUN, Xiaoxun; and KOENIG, Sven. Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems. (2011). *AAMAS 2011: Proceedings of 10th International Conference on Autonomous Agents and Multiagent Systems: May 2-6, 2011, Taipei, Taiwan*. 1069-1070. Research Collection School Of Information Systems.
Available at: http://ink.library.smu.edu.sg/sis_research/1343

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems

William Yeoh[†], Pradeep Varakantham[‡], Xiaoxun Sun[⋆], and Sven Koenig[+]
[†]Department of Computer Science, New Mexico State University
[‡]School of Information Systems, Singapore Management University
[⋆]Google Inc.
[+]Department of Computer Science, University of Southern California
[†]wyeoh@cs.nmsu.edu [‡]pradeepv@smu.edu.sg [⋆]xiaoxunsun@google.com [+]skoenig@usc.edu

***Abstract*—Distributed constraint optimization (DCOP) problems are well-suited for modeling multi-agent coordination problems. However, it only models static problems, which do not change over time. Consequently, researchers have introduced the Dynamic DCOP (DDCOP) model to model dynamic problems. In this paper, we make two key contributions: (a) a procedure to reason with the incremental changes in DDCOPs and (b) an incremental pseudo-tree construction algorithm that can be used by DCOP algorithms such as any-space ADOPT and any-space BnB-ADOPT to solve DDCOPs. Due to the incremental reasoning employed, our experimental results show that any-space ADOPT and any-space BnB-ADOPT are up to 42% and 38% faster, respectively, with the incremental procedure and the incremental pseudo-tree reconstruction algorithm than without them.**

I. Introduction

Distributed constraint optimization (DCOP) problems are problems where agents need to coordinate their value assignments to minimize the sum of the resulting constraint costs [1], [2], [3], [4], [5]. They are well-suited for modeling multi-agent coordination problems where the primary interactions are between local subsets of agents, such as the scheduling of meetings [6], the coordination of sensors in networks [7], and the generation of coalition structures [8]. Unfortunately, DCOP problems only model static problems or, in other words, problems that do not change over time. In the above mentioned coordination problems, various events that change the problem can occur. For example, in the scheduling of meetings, new meetings might need to be scheduled, time constraints of meeting participants could have changed, or the priorities of meetings could have changed.

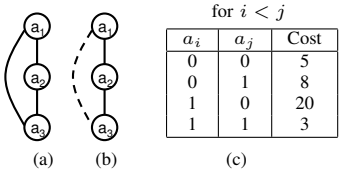
As a result, researchers have introduced the Dynamic DCOP (DDCOP) model [9], [10], [11], [12], [13], which is modeled as a sequence of DCOPs, each partially different from the DCOP preceding it. Existing algorithms thus far take a *reactive* approach and solves dynamic DCOPs by searching for a new solution each time the problem changes. Since the change between subsequent DCOPs can be small, it might be possible to reuse information from previous DCOPs to speed up the search of the current DCOP. In this paper, we investigate how to maximize the amount of information reuse by DCOP search algorithms. Specifically, we introduce an incremental procedure and an incremental pseudo-tree reconstruction algorithm that can be used by any-space DCOP search algorithms to reuse information gained from solving the previous DCOP problem. Our experimental results show that any-space ADOPT [14] and any-space BnB-ADOPT [15] are up to 42% and 38% faster, respectively, with the incremental procedure and incremental pseudo-tree reconstruction algorithm than without them.

II. Background

A. DCOP Problems

A DCOP problem P is defined as a tuple $\langle A, D, F \rangle$. $A = \{a_i\}_0^n$ is the finite set of agents. $D = \{d_i\}_0^n$ is the set of finite domains, where domain d_i is the set of possible values for agent $a_i \in A$. $F = \{f_i\}_0^m$ is the set of binary constraints, where each constraint $f_i : d_{i_1} \times d_{i_2} \rightarrow \mathbb{R}^+ \cup \infty$ specifies its non-negative constraint cost as a function of the values of the two different agents $a_{i_1}, a_{i_2} \in A$ that share the constraint. Each agent is responsible for assigning itself values from its domain. The agents coordinate their value assignments by exchanging messages. A solution is an agent-value assignment for a subset of agents. Its cost is the sum of the constraint costs of all constraints shared by agents with known values. A solution is called *complete* iff it is an agent-value assignment for all agents. Solving a DCOP problem means to find a cost-minimal complete solution.

DCOP problems are commonly visualized as *constraint graphs*, whose vertices are the agents and whose edges are the constraints. Most DCOP search algorithms operate on *pseudo-trees*. Pseudo-trees are spanning trees of fully connected constraint graphs such that no two vertices in different subtrees of the spanning tree are connected by an edge in the constraint graph. Figure 1(a) shows the constraint graph of an example DCOP problem with three agents that can each take on the values 0 or 1, Figure 1(b) shows one possible pseudo-tree (the dotted line is called a *backedge*, which is an edge of the constraint graph that does not connect a pair of parent-child nodes), and Figure 1(c) shows the constraint costs. Sibling subtrees of pseudo-trees represent independent subproblems in a DCOP problem. DCOP algorithms prefer pseudo-trees with many sibling subtrees or, equivalently, pseudo-trees with small depths since they can solve independent subproblems in parallel and thus solve them faster.



for $i < j$

a_i	a_j	Cost
0	0	5
0	1	8
1	0	20
1	1	3

(c)

Fig. 1. Example DCOP Problem

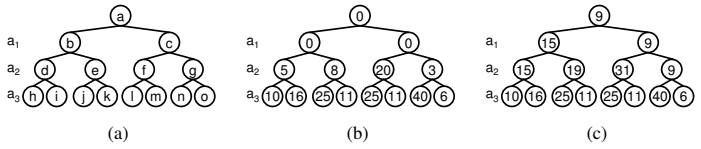


Fig. 2. Search Trees

B. ADOPT and BnB-ADOPT

ADOPT [1] and BnB-ADOPT [4] transform the constraint graph to a pseudo-tree in a pre-processing step and then search for a cost-minimal complete solution. They have identical memory requirements and communication frameworks but have different search strategies.

We use search trees and terminology from A* to visualize the operation of ADOPT and BnB-ADOPT. Each level of a search tree corresponds to an agent. For our example DCOP problem, the level of depth 1 corresponds to agent a_1. A left branch that enters a level means that the corresponding agent takes on the value 0. A right branch means that the corresponding agent takes on the value 1. A node in the search tree thus corresponds to a solution. For our example DCOP problem, the solution of node j is $(a_1 = 0, a_2 = 1, a_3 = 0)$, where we use the identifiers shown in Figure 2(a) to refer to nodes. The numbers in the nodes in Figure 2(b) are their delta costs. The *delta cost* of a node is the sum of the costs of all constraints in the solution of that node that involve the agent corresponding to the level of that node. For example, the delta cost of node j is the sum of the costs of all constraints in the solution of node j that involve agent a_3. Therefore, it is the sum of the cost of the constraint between agents a_1 and a_3, which is five, and the cost of the constraint between agents a_2 and a_3, which is twenty, resulting in twenty five. The numbers in the nodes in Figure 2(c) are their gamma costs. The *gamma cost* of a node is the smallest sum of the delta costs of all nodes along any branch of the search tree from that node to a leaf node. The gamma cost of the root node is thus the cost of a cost-minimal complete solution.

ADOPT and BnB-ADOPT are different in their choice of search strategy for traversing the search tree. ADOPT uses *best-first search* while BnB-ADOPT uses *depth-first branch-and-bound search*. ADOPT and BnB-ADOPT are similar in that each agent maintains only the lower and upper bounds on the gamma costs of at most two nodes in its level of the search tree at all times due to memory limitations. They are the children of the node whose solution is the agent's *current context*, which is the agent's assumption on the agent-value assignments for all its ancestors in the pseudo-tree. For example, if the current context of agent a_3 is $(a_1 = 0, a_2 = 0)$, then it maintains the lower and upper bounds on the gamma costs of nodes h and i. Therefore, each agent can maintain only *one* information unit at all times. An *information unit* consists of the lower and upper bounds that the agent maintains for one context. On the other hand, each agent in any-space ADOPT [14] and any-space BnB-ADOPT [15] can maintain *multiple* information units, where each information unit consists of the lower and upper bounds for a different context. Runtimes of any-space ADOPT and any-space BnB-ADOPT typically decrease as they have more memory available [15].

Finally, researchers have also further optimized BnB-ADOPT by (1) reducing the number of messages that it requires [16]; (2) extending it to maintain soft arc consistency during search [17], [18], [19]; and (3) combined it with ADOPT to form a generalized asynchronous DCOP search algorithm [20].

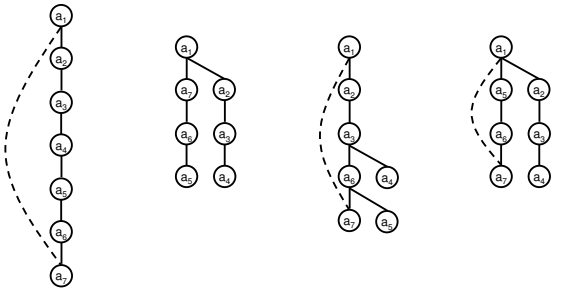
C. DDCOP Problems

A Dynamic DCOP (DDCOP) problem is defined by a tuple $\langle P_0, T, C, \Delta \rangle$. P_0 is the DCOP problem that models the initial coordination problem. T is the time horizon, where each time step $1 \leq t \leq T$ represents a point in time where the coordination problem changes. In this paper, we assume that the time between subsequent time steps are sufficiently large to solve the current problem. $C = \{c_i\}_0^k$ is the set of possible changes and $\Delta = \{\Delta_t\}_1^T$ captures the dynamism in the coordination problem by representing the changes at each time step. $\Delta_t = \{P(c_i)|c_i \in C\}$ is the probability distribution of the changes that can occur at time step t.

We consider a DDCOP problem as a sequence of DCOP problems with changes between them. Solving a DDCOP problem optimally means finding a cost-minimal solution for each DCOP problem in the sequence. Therefore, this approach is a *reactive* approach since it does not consider future changes. The advantage of this approach is that solving DDCOP problems is no harder than solving T DCOP problems. Researchers have used this approach to solve DDCOPs [9], where they introduce a super-stabilizing DPOP algorithm that is able to reuse information from previous DCOPs to speed up the search for the solution for the current DCOP. Our approach is similar except that (1) we investigate search-based algorithms instead of inference based algorithms and (2) we investigate ways to maximize the amount of information reuse through pseudo-tree reconstruction.

Alternatively, a *proactive* approach predicts future changes in the DDCOP problem and finds robust solutions that require little or no changes despite future changes. Researchers have used this approach to solve dynamic constraint satisfaction problems [23], [24] but not DDCOP problems to the best of our knowledge.

Researchers have also proposed other models for DDCOP problems including a model where agents have deadlines to choose their values and incur a cost for changing their value assignments when the problem changes [10] and a



(a) Initial pseudo-tree (b) DFS pseudo-tree (c) Mobed pseudo-tree (d) HARP pseudo-tree

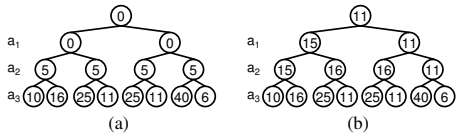
Fig. 3. Pseudo-trees

model where agents can have imperfect knowledge about their environment [11]. Lastly, researchers have also introduced incomplete algorithms to solve DDCOP problems suboptimally [25], [26].

D. Pseudo-tree Algorithms

One can use DCOP algorithms to solve DDCOP problems by running the algorithm each time the problem changes. However, before solving the current DCOP problem, DCOP algorithms may need to reconstruct their pseudo-trees to reflect the changes from the previous DCOP problem. For example, if an agent is removed, then a DCOP algorithm needs to remove that agent from its pseudo-tree. Since constructing optimal pseudo-trees is NP-hard [21], researchers have developed greedy distributed algorithms to construct them. For example, the distributed Depth-First Search (DFS) algorithm constructs the pseudo-tree for each DCOP problem from scratch by using DFS to traverse the constraint graph for that DCOP problem [22]. Figure 3(a) shows an example initial pseudo-tree, and Figure 3(b) shows the pseudo-tree reconstructed by the distributed DFS algorithm after the constraint between agents a_4 and a_5 is removed.

Another example is the Multiagent Organization with Bounded Edit Distance (Mobed) algorithm, which constructs pseudo-trees with small edit distances between subsequent DCOP problems [12]. The edit distance between two pseudo-trees is the smallest number of parent-child relationships that must be re-assigned, added or deleted in order for both pseudo-trees to become isomorphic. It requires a different algorithm, such as the distributed DFS algorithm, to construct the pseudo-tree of the first DCOP problem. For each new agent that is added to the DCOP problem, Mobed identifies an insertion point in the pseudo-tree of the previous DCOP problem and adds the new agent to the pseudo-tree at that insertion point. For each agent that is removed from the DCOP problem, Mobed removes that agent from the pseudo-tree and makes all children of the removed agent the children of the parent of the removed agent. For each constraint that is added or removed, Mobed removes and adds all agents that share that constraint. For example, Figure 3(c) shows the pseudo-tree reconstructed by the Mobed algorithm after the constraint between agents a_4 and a_5 was removed.



(a) (b)

Fig. 4. New Delta and Gamma Costs

III. REUSEBOUNDS PROCEDURE

When solving a DDCOP problem, instead of solving each DCOP problem from scratch, any-space DCOP search algorithms can reuse information from searches of previous DCOP problems to guide their search to potentially solve the current DCOP problem faster. Therefore, we introduce the ReuseBounds procedure, an incremental procedure that any-space ADOPT and any-space BnB-ADOPT can use to identify lower and upper bounds obtained from solving the previous DCOP problem that can be reused to solve the current DCOP problem. After each change in the DDCOP problem, any-space ADOPT and any-space BnB-ADOPT first reconstruct their pseudo-trees and then call the ReuseBounds procedure before solving the current DCOP problem.

Initially, the ReuseBounds procedure identifies potentially affected and unaffected agents in the current DCOP problem. The *potentially affected agents* are those agents whose lower and upper bounds from the previous DCOP problem may no longer be correct bounds for the current DCOP problem. The *unaffected agents* are the other agents. Once the agents are identified, each potentially affected agent purges all its cached information units before solving the current DCOP problem. Unaffected agents reuse the lower and upper bounds in all their cached information units.

Specifically, an agent is a potentially affected agent iff the gamma costs of nodes in its level of the search tree can change between the previous and current DCOP problems. A gamma cost can change iff one can assign some costs to each constraint that is added, removed, or whose costs changed in such a way that the gamma cost indeed changes. If the gamma cost of a node changes, then the lower and upper bounds of that node, which were correct bounds in the previous DCOP problem, may no longer be correct bounds in the current DCOP problem. Unfortunately, an agent cannot directly discern if the gamma costs of nodes in its level of the search tree can change because it needs to solve the current DCOP problem to do so. However, an agent can determine if it is a potentially affected agent by checking if it has one or more of the following properties:

- **Property 1:** The agent shares an added constraint, deleted constraint, or constraint with changed constraint costs with another agent. If the agent shares the constraint with a descendant, then it is a potentially affected agent (see Property 3). If the agent shares the constraint with an ancestor, then the changes in the constraints can change the delta costs of some nodes in its level of the search tree, which in turn can change the gamma costs of those nodes. The agent is thus a potentially affected agent.

- **Property 2:** The agent has different children between the previous and current DCOP problems. This difference can change the children of nodes in the level of the agent in the search tree, which in turn can change the gamma costs of those nodes. The agent is thus a potentially affected agent.
- **Property 3:** The agent a has a descendant that is a potentially affected agent. If the descendant is a potentially affected agent, then the sum of the delta costs of all nodes along some branch of the search tree from some leaf node to some node in the level of the descendant can change. This branch of the search tree can be extended up the search tree to some node in the level of agent a in the search tree. Therefore, the sum of the delta costs of all nodes along this extended branch can also change, which in turn can change the gamma costs of the node (in that branch) that is in the level of agent a. Agent a is thus a potentially affected agent.

For example, assume that the costs of the constraint between agents a_1 and a_2 change to five for all value combinations in our example DCOP problem. Figure 4(a) shows the new delta costs, and Figure 4(b) shows the new gamma costs. The change in the constraint costs changes the delta costs of nodes d through g, which in turn changes the gamma costs of those nodes. Thus, agent a_2 is a potentially affected agent since it maintains the lower and upper bounds of those nodes (Property 1). The change in the delta costs also changes the gamma costs of nodes a through c. Thus, agent a_1 is a potentially affected agent since it maintains the lower and upper bounds of those nodes (Property 3).

Each agent can directly discern if it has Properties 1 or 2. However, it cannot directly discern if it has Property 3 since it knows neither the constraints nor the set of children of each of its descendants. Therefore, each agent in any-space ADOPT and any-space BnB-ADOPT runs the ReuseBounds procedure to discern if it has Property 3 after it reconstructs the pseudo-tree for the current DCOP problem. The idea behind the procedure is the following. Leaf agents first identify themselves as potentially affected or unaffected agents, which they can do because they do not need to consider Property 3 since they have no descendants. The leaf agents then send RESPONSE messages to their parents with that information. The parents then identify themselves as potentially affected or unaffected agents, which they can now do because they know whether they have Property 3 or not or, in other words, whether any of their children are potentially affected agents or not. The agents propagate the RESPONSE messages up the pseudo-tree until they reach the root agent. Ideally, the leaf agents ought to automatically start the process and send RESPONSE messages once there is a change in the problem. Unfortunately, they are not aware of all possible changes in the problem since they are only aware of their constraints and the agents that they are constrained with. Therefore, the ReuseBounds procedure use START and QUERY messages to trigger the leaf agents to send RESPOND messages.

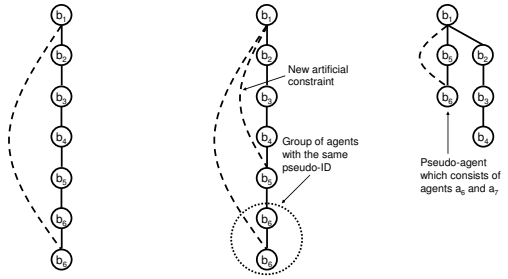
Figure 5 shows the pseudocode. The code is identical for every agent with variable a pointing to the agent executing the code. Each agent initially identifies itself as an unaffected

```
01 procedure ReuseBounds()
02 sentSTART := amAffected := false;
03 loop forever
04   if(message queue is not empty)
05     pop msg off message queue;
06     When Received(msg);
07   if(!sentSTART and detected changes in its constraints
     or its set of children)
08     sentSTART := true;
09     Send(START) to parent;
10     Send(QUERY) to each child if a is root;

11 procedure When Received(START)
12 if(!sentSTART)
13   sentSTART := true;
14   Send(START) to parent;
15   Send(QUERY) to each child if a is root;

16 procedure When Received(QUERY)
17 Send(QUERY) to each child;
18 if(a is a leaf)
19   if(detected changes in its constraints or its set of children)
20     amAffected := true;
21   Send(RESPONSE, a, amAffected) to parent;

22 procedure When Received(RESPONSE, c, c.amAffected)
23 if(c.amAffected or detected changes in its constraints or its set of children)
24   amAffected := true;
25 if(received a RESPONSE message from each child)
26   Send(RESPONSE, a, amAffected) to parent;
27   if(a is root)
28     if(amAffected)
29       purge all information units;
30     Send(STOP) to each child;
31     restart DCOP algorithm to solve the current DCOP problem;

32 procedure When Received(STOP)
33 if(amAffected)
34   purge all information units;
35 Send(STOP) to each child;
36 restart DCOP algorithm to solve the current DCOP problem;
```

Fig. 5. Pseudocode of ReuseBounds

agent [Line 2]. Then, each agent with Properties 1 or 2 sends a START message to its parent [Lines 7-9], which is propagated up the pseudo-tree to trigger the root agent to send QUERY messages [Lines 11-14]. When the root agent receives a START message, it sends a QUERY message to each of its children [Line 15], which is propagated down the pseudo-tree to trigger the leaf agents to send RESPOND messages [Lines 16-17]. When a leaf agent receives a QUERY message, it identifies itself as a potentially affected agent if it has Properties 1 or 2 [Lines 18-20] and then sends a RESPOND message with that information to its parent [Line 21]. When an agent receives a RESPOND message from each of its children, it identifies itself as a potentially affected agent if it has Properties 1, 2, or 3 [Lines 22-24] and sends a RESPOND message with that information to its parent [Lines 25-26]. (An agent has Property 3 if it receives a RESPOND message from a potentially affected agent.) Therefore, RESPOND messages propagate up the pseudo-tree until they reach the root agent. Finally, when the root agent receives a RESPOND message from each of its children, each agent with Properties 1, 2, or 3 must have identified itself as a potentially affected agent. The root agent thus sends a STOP message to each of its children [Line 30], which is propagated down the pseudo-tree and ends the ReuseBounds procedure [Lines 32-36].

After running the ReuseBounds procedure, any-space ADOPT and any-space BnB-ADOPT restart as usual to solve



(a) Assigning pseudo-IDs (b) Artificially constraining agents b_1 and b_5 (c) Constructing DFS pseudo-tree

Fig. 6. HARP Pseudo-tree Reconstruction Steps

the current DCOP problem except that the unaffected agents now reuse their bounds from the previous DCOP problem in the current DCOP problem. Therefore, any-space ADOPT and any-space BnB-ADOPT might be able to solve the current DCOP problem faster since the unaffected agents do not need to recompute the lower and upper bounds that they reused.

IV. HARP Algorithm

The Mobed algorithm constructs pseudo-trees with small edit distances. On the other hand, the distributed DFS algorithm constructs its pseudo-trees from scratch, which can have large edit distances. Therefore, there is usually a larger number of unaffected agents in the Mobed pseudo-trees than in the DFS pseudo-trees. For example, in our example pseudo-trees of Figure 3, there is one unaffected agent (= agent a_7) in the Mobed pseudo-tree and there are no unaffected agents in the DFS pseudo-tree. On the other hand, DFS pseudo-trees can have smaller depths than Mobed pseudo-trees. For example, in our example pseudo-trees of Figure 3, the depth of the DFS pseudo-tree is one smaller than the depth of the Mobed pseudo-tree. Pseudo-trees with smaller depths are desirable since they have a larger number of independent subtrees or, synonymously, a larger number of independent subproblems. Thus, there is a tradeoff between the depths and edit distances of pseudo-trees.

We therefore introduce the *Hybrid Algorithm for Reconstructing Pseudo-trees* (HARP), an incremental pseudo-tree reconstruction algorithm that reuses parts of the pseudo-tree of the previous DCOP problem to construct the pseudo-tree of the current DCOP problem. It combines the principles and strengths of the Mobed and distributed DFS algorithms. Like the Mobed algorithm, HARP aims to preserve the parent-child relationships of unaffected agents in the pseudo-tree and, like the distributed DFS algorithm, HARP reconstructs the part of the pseudo-tree with the affected agents from scratch. *Affected agents* are agents from the previous DCOP problem that are *guaranteed* to be potentially affected agents in the current DCOP problem regardless of the choice of pseudo-tree reconstruction algorithm. They are (1) the agents that share an added constraint, deleted constraint or constraint with changed costs with another agent (Property 1) and (2) their ancestors in

```
01 procedure HARP()
02 sentSTART := amAffected := false;
03 pseudoID := agentID;
04 loop forever
05   if(message queue is not empty)
06     pop msg off message queue;
07     When Received(msg);
08   if(!sentSTART and detected changes in its constraints)
09     sentSTART := true;
10     Send(START) to parent;
11     Send(QUERY) to each child if a is root;

12 procedure When Received(START)
13 if(!sentSTART)
14   sentSTART := true;
15   Send(START) to parent;
16   Send(QUERY) to each child if a is root;

17 procedure When Received(QUERY)
18 Send(QUERY) to each child;
19 if(a is a leaf)
20   if(detected changes in its constraints)
21     amAffected := true;
22   Send(RESPONSE, a, amAffected) to parent;

23 procedure When Received(RESPONSE, c, c.amAffected)
24 if(c.amAffected or detected changes in its constraints)
25   amAffected := true;
26 if(received a RESPONSE message from each child)
27   Send(RESPONSE, a, amAffected) to parent;
28   Send(PSEUDOID, pseudoID, amAffected) to each child if a is root;
29 procedure When Received(PSEUDOID, p.pseudoID, p.amAffected)
30 if(!amAffected and p.amAffected)
31   Send(CONSTRAINT, a, sep(a)) to each ancestor in sep(a);
32 else
33   if(!p.amAffected)
34     pseudoID := p.pseudoID;
35   Send(PSEUDOID, pseudoID, amAffected) to each child;
36   Send(PSEUDOID-ACK) to parent if a is leaf;

37 procedure When Received(PSEUDOID-ACK)
38 if(received a PSEUDOID-ACK message from each child)
39   Send(PSEUDOID-ACK) to parent;
40 if(a is root)
41   Send(STOP) to each child;
42   run distributed DFS algorithm;

43 procedure When Received(CONSTRAINT, c, c.SCP)
44 artificially constrain itself with each agent in c.SCP;
45 Send(CONSTRAINT-ACK) to c;

46 procedure When Received(CONSTRAINT-ACK)
47 if(received a CONSTRAINT-ACK message from each ancestor in sep(a))
48   Send(PSEUDOID, pseudoID, amAffected) to each child;
49   Send(PSEUDOID-ACK) to parent if a is leaf;

50 procedure When Received(STOP)
51 Send(STOP) to each child;
52 run distributed DFS algorithm;
```

Fig. 7. Pseudocode of HARP

the pseudo-tree of the previous DCOP problem (Property 3). (Other agents might become potentially affected agents as well but that depends on the choice of pseudo-tree reconstruction algorithm and is thus not guaranteed.)

HARP operates on the pseudo-tree of the previous DCOP problem to identify the affected agents and calls the distributed DFS algorithm to construct the pseudo-tree of the current DCOP problem in such a way that non-affected agents are unaffected agents in the current DCOP problem. Recall that unaffected agents are agents that are not potentially affected agents. For example, imagine that the constraint between agents a_4 and a_5 of Figure 3(a) is removed. The affected agents are agents a_4 and a_5 since they have Property 1 and

agents a_1, a_2 and a_3 since they have Property 3. Figure 6 shows the steps of HARP to reconstruct the pseudo-tree for the current DCOP problem. HARP assigns a unique pseudo-ID to all agents in the problem with the exception that, all agents in an unaffected subtree are assigned the same pseudo-ID. An *unaffected subtree* is a strict subtree that has only unaffected agents. In our example, HARP assigns agents a_1, a_2, a_3, a_4 and a_5 the pseudo-IDs b_1, b_2, b_3, b_4 and b_5, respectively. HARP assigns agents a_6 and a_7 the same pseudo-ID b_6 since they are in an unaffected subtree. Figure 6(a) shows the pseudo-tree with the pseudo-ID of each agent. For each group of agents with the same pseudo-ID that is not in a larger group of agents with the same pseudo-ID, HARP artificially constrains all their parents and pseudo-parents that are not in the group to each other. An agent is a *pseudo-parent* of another agent if the former agent is an ancestor of the latter agent in the pseudo-tree and they are connected via a backedge in the pseudo-tree. In our example, HARP artificially constrains agents a_1 (with pseudo-ID b_1) and a_5 (with pseudo-ID b_5) since agent a_1 is a pseudo-parent of agent a_7 and agent a_5 is a parent of agent a_6. Figure 6(b) shows the pseudo-tree with the new artificial constraint.

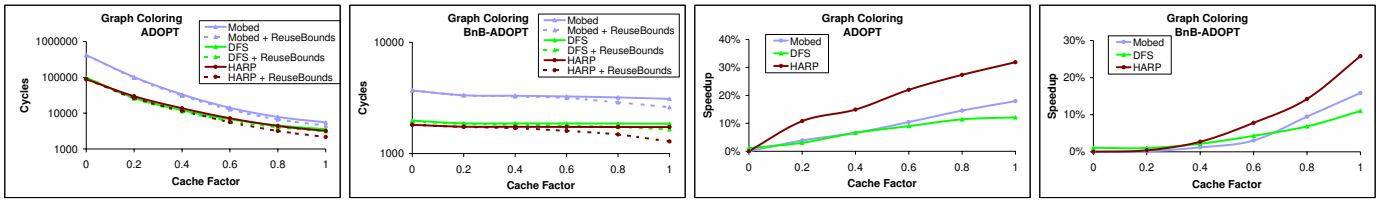
HARP then runs the distributed DFS algorithm except that it treats all agents with the same pseudo-ID as a single pseudo-agent and sets pseudo-agents as children of the current agent only if every affected agent that is constrained with the current agent is already in the pseudo-tree. Therefore, pseudo-agents are leaf agents in the new pseudo-tree because every agent that they are constrained with was chosen first due to the artificial constraints. Therefore, non-affected agents do not have Properties 2 and 3 because pseudo-agents are leaf agents and all agents in a pseudo-agent thus have the same descendants, which are non-affected agents themselves, in the new pseudo-tree. Non-affected agents do not have Property 1 either because they are affected agents otherwise. Therefore, all non-affected agents are unaffected agents in the current DCOP problem. Figure 6(c) shows the resulting pseudo-tree with pseudo-agents in our example, and Figure 3(d) shows the pseudo-tree with actual agents. The HARP pseudo-tree has the same depth as the DFS pseudo-tree and has two unaffected agents (= agents a_6 and a_7) compared to one unaffected agent in the Mobed pseudo-tree and no unaffected agents in the DFS pseudo-tree.

Figure 7 shows the pseudocode of the HARP algorithm. The code is identical for every agent with variable a pointing to the agent executing the code and $sep(a)$ is the separator set of variable a, i.e., the set of its ancestors that are parents or pseudo-parents of its descendants or the agent itself. There are two phases in the HARP algorithm. The first phase identifies affected agents and is very similar to how the ReuseBounds procedure identifies potentially affected agents. The second phase assigns pseudo-IDs to agents and imposes artificial constraints between agents before calling the distributed DFS algorithm to reconstruct the pseudo-tree. In general, the first phase of the HARP algorithm is as follows: Agents with Property 1 sends a START message to its parent [Lines 8-9], which is propagated up the pseudo-tree to trigger the root agent to send QUERY messages [Lines 12-15]. When the root agent receives a START message, it sends a QUERY message to each of its children [Line 16], which is propagated down the pseudo-tree to trigger the leaf agents to send RESPOND messages [Lines 17-18]. When a leaf agent receives a QUERY message, it identifies itself as an affected agent if it has Property 1 [Lines 19-21] and then sends a RESPOND message with that information to its parent [Line 22]. When an agent receives a RESPOND message, it identifies itself as an affected agent if it has Properties 1 or 3 [Lines 23-25] and sends a RESPOND message with that information to its parent [Line 27]. (An agent has Property 3 if it receives a RESPOND message from an affected agent.) Therefore, RESPOND messages propagate up the pseudo-tree until they reach the root agent. When the root agent receives a RESPOND message from each of its children, each agent with Properties 1 or 3 must have identified itself as an affected agent. The root agent thus starts the second phase of the HARP algorithm by sending a PSEUDOID message to each of its children [Line 28], which is propagated down the pseudo-tree. When an agent receives a PSEUDOID message, if it is an affected agent or is the root of an unaffected subtree, then it sets its pseudo-ID to its unique agent-ID. Otherwise, it sets its pseudo-ID to the pseudo-ID of the root of the unaffected subtree that it is in [Lines 32-33]. Additionally, when a root agent a of an unaffected subtree receives a PSEUDOID message, it sends a CONSTRAINT message to each of its ancestors in $sep(a)$ [Lines 29-31] such that they artificially constrain themselves to each other [Lines 43-44]. When a leaf agent receives a PSEUDOID message, it sends a PSEUDOID-ACK message to its parent [Line 36], which is propagated up the pseudo-tree [Lines 37-39]. Finally, when the root agent receives a PSEUDOID-ACK message from each of its children, it sends STOP messages to its children [Lines 40-41], which is propagated down the pseudo-tree and ends the HARP algorithm [Lines 50-52].

V. Experimental Results

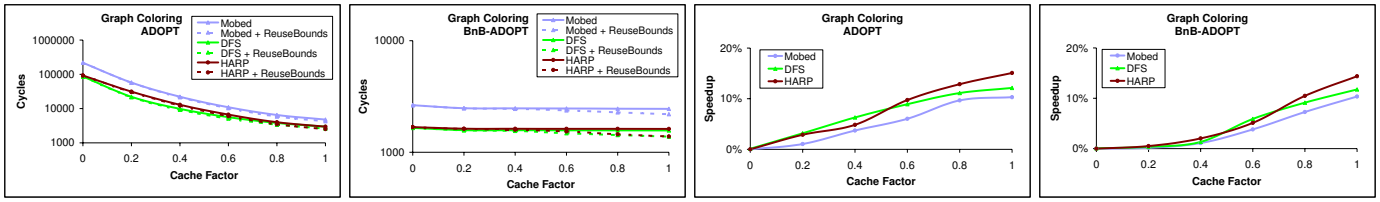
We now compare any-space ADOPT and any-space BnB-ADOPT using the ReuseBounds procedure and the distributed DFS, Mobed, and HARP pseudo-tree reconstruction algorithms. We measure the runtimes in cycles [1]. Although we do not measure the runtimes in other metrics, such as NCCCs [27], we believe that the reported trends carry over to those metrics as well because the number of constraint checks and the number of messages sent by each agent do not vary much across cycles. We vary the amount of memory of each agent with the cache factor metric [28] from 0 (each agent can cache only one information unit) to 1 (each agent can cache all information units). We use the MaxEffort and MaxPriority caching schemes [15] for any-space ADOPT and any-space BnB-ADOPT, respectively.

We consider the following five types of changes in our experiments: c_1 is the change in the costs of a random constraint, c_2 is the removal of a random constraint, c_3 is the addition of a random constraint, c_4 is the removal of



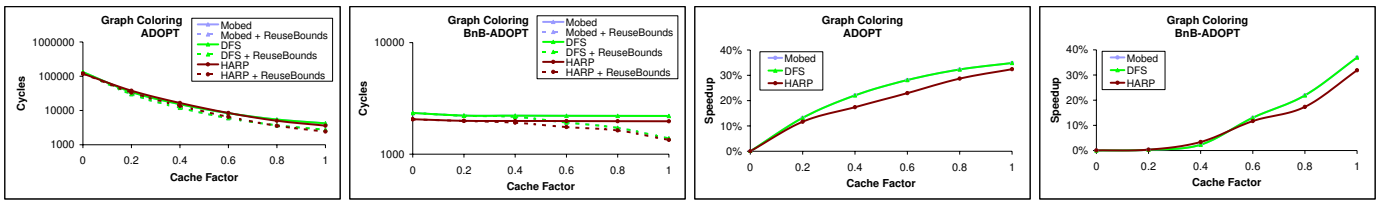
(a) (b) (c) (d)

Fig. 8. Experimental Results for Type 1 DDCOP Problems



(a) (b) (c) (d)

Fig. 9. Experimental Results for Type 2 DDCOP Problems



(a) (b) (c) (d)

Fig. 10. Experimental Results for Type 3 DDCOP Problems

a random agent, and c_5 is the addition of a random agent. We randomly generated graph coloring problems of density 2, domain cardinality 5 and constraint costs of 0–10,000 as the initial DCOP problems P_0. We constructed three types of DDCOP problems with time horizon $T = 5$ and one change at each time step:

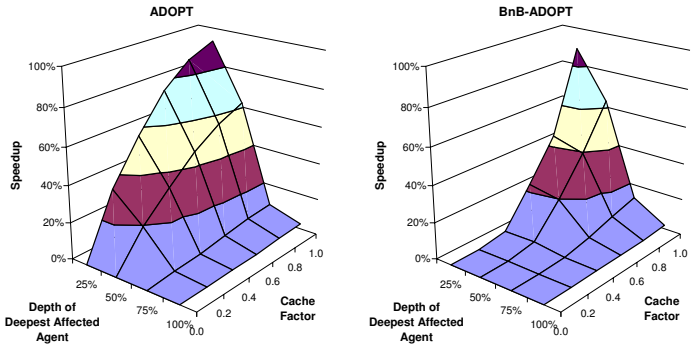
- **Type 1**: $\Delta_t = \{P(c_t) = 1\}$,
- **Type 2**: $\Delta_t = \{P(c_i) = P(c_j) \mid c_i, c_j \notin \bigcup_{1 \leq t' < t} \mathbb{C}_{t'}\}$, where $\mathbb{C}_t$ is the change that occurred at time step t,
- **Type 3**: $\Delta_t = \{P(c_1) = 1\}$.

We average the experimental results over 50 DDCOP problem instances of each type.

Figures 8, 9, and 10 show our experimental results for any-space ADOPT and any-space BnB-ADOPT with the ReuseBounds procedure and the pseudo-tree reconstruction algorithms on Types 1, 2, and 3 DDCOP problems, respectively. We make the following observations:

- Figures 8(a,b), 9(a,b), and 10(a,b) show that the runtimes of any-space ADOPT and any-space BnB-ADOPT decrease as their cache factors increase. The reason for this behavior is that they need to expand fewer nodes when they cache more information [15].
- The runtimes of any-space ADOPT and any-space BnB-ADOPT are smaller with the ReuseBounds procedure than without the procedure. Figures 8(c,d), 9(c,d), and 10(c,d) show the speedups gained with the ReuseBounds procedure. We calculate the speedup by taking the difference in the runtimes with and without the ReuseBounds procedure and normalizing it by the runtime without the ReuseBounds procedure. The figures show that the speedup increases as the cache factor increases for all three pseudo-tree reconstruction algorithms. The reason for this behavior is that the unaffected agents can cache and reuse more lower and upper bounds from the previous DCOP problems as the cache factor increases.
- The runtimes of any-space ADOPT and any-space BnB-ADOPT with the DFS and Mobed algorithms are identical for each cache factor for Type 3 DDCOP problems. Mobed does not reconstruct the pseudo-tree for the current DCOP problem when the only change is the change in constraint costs. DFS reconstructs the exact same pseudo-tree for the current DCOP problem as the pseudo-tree for the previous DCOP problem since all the agents are constrained in the exact same way for both DCOP problems.
- Any-space ADOPT and any-space BnB-ADOPT with the HARP algorithm and the ReuseBounds procedure are up to 42% and 38% faster, respectively, than with the distributed DFS algorithm and without the ReuseBounds procedure.

To better understand the sources of speedup, we performed an additional experiment with DDCOP problems with time horizon $T = 1$ and $\Delta_1 = \{P(c_1) = 1\}$, where we varied the depth of the deepest affected agent in the pseudo-tree. Figure 11 shows the experimental results for any-space ADOPT and any-space BnB-ADOPT using the distributed DFS algorithm. (The results with the HARP algorithm are similar.) They show that the speedup increases as the depth



(a) (b)

Fig. 11. Secondary Experimental Results

of the deepest affected agent decreases. The reason for this behavior is that the number of unaffected agents increases as the depth of the deepest affected agent decreases, and more lower and upper bounds from the previous DCOP problem are thus reused. Overall, the ReuseBounds procedure experimentally speeds up any-space ADOPT and any-space BnB-ADOPT for all three pseudo-tree reconstruction algorithms, and the speedup increases as the cache factor increases and as the depth of the deepest affected agent decreases.

VI. Conclusions

DDCOP problems are well-suited for modeling multi-agent coordination problems that change over time. In this paper, we introduced the ReuseBounds procedure and the HARP pseudo-tree reconstruction algorithm that can be employed by any-space ADOPT and any-space BnB-ADOPT to solve DDCOP problems optimally. The ReuseBounds procedure allows the agents that are unaffected by the changes in the DDCOP problem to reuse their lower and upper bounds from the previous DCOP problem. The HARP pseudo-tree reconstruction algorithm reconstructs the pseudo-tree for the current DCOP problem in such a way that maximizes the number of such unaffected agents. Our experimental results show that the speedup gained by using the ReuseBounds procedure increases as the amount of available memory increases. In general, we expect the ReuseBounds procedure to apply to other DCOP search algorithms with other pseudo-tree reconstruction algorithms as well since all DCOP search algorithms maintain lower and upper bounds on the solution costs. We also expect the HARP algorithm to apply to other DCOP algorithms that operate on pseudo-trees.

Acknowledgments

This material is based upon work supported by NSF, first while Sven Koenig was serving at NSF and later under grant numbers 1409987, 1319966, and 1345232. It is also based upon work supported by ARL/ARO under contract/grant number W911NF-08-1-0468, ONR in form of a MURI under contract/grant number N00014-09-1-1031, and DOT under contract/grant number DTFH61-11-C-00010.

References

[1] P. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "ADOPT: Asynchronous distributed constraint optimization with quality guarantees," *Artificial Intelligence*, vol. 161, no. 1–2, pp. 149–180, 2005.

[2] A. Petcu and B. Faltings, "A scalable method for multiagent constraint optimization," in *Proceedings of IJCAI*, 2005, pp. 1413–1420.

[3] A. Gershman, A. Meisels, and R. Zivan, "Asynchronous Forward-Bounding for distributed COPs," *Journal of Artificial Intelligence Research*, vol. 34, pp. 61–88, 2009.

[4] W. Yeoh, A. Felner, and S. Koenig, "BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm," *Journal of Artificial Intelligence Research*, vol. 38, pp. 85–133, 2010.

[5] W. Yeoh and M. Yokoo, "Distributed problem solving," *AI Magazine*, vol. 33, no. 3, pp. 53–65, 2012.

[6] R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham, "Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling," in *Proceedings of AAMAS*, 2004, pp. 310–317.

[7] A. Farinelli, A. Rogers, A. Petcu, and N. Jennings, "Decentralised coordination of low-power embedded devices using the Max-Sum algorithm," in *Proceedings of AAMAS*, 2008, pp. 639–646.

[8] S. Ueda, A. Iwasaki, and M. Yokoo, "Coalition structure generation based on distributed constraint optimization," in *Proceedings of AAAI*, 2010, pp. 197–203.

[9] A. Petcu and B. Faltings, "Superstabilizing, fault-containing multiagent combinatorial optimization," in *Proceedings of AAAI*, 2005, pp. 449–454.

[10] ——, "Optimal solution stability in dynamic, distributed constraint optimization," in *Proceedings of IAT*, 2007, pp. 321–327.

[11] R. Lass, E. Sultanik, and W. Regli, "Dynamic distributed constraint reasoning," in *Proceedings of AAAI*, 2008, pp. 1466–1469.

[12] E. Sultanik, R. Lass, and W. Regli, "Dynamic configuration of agent organizations," in *Proceedings of IJCAI*, 2009, pp. 305–311.

[13] R. Zivan, H. Yedidsion, S. Okamoto, R. Glinton, and K. Sycara, "Distributed constraint optimization for teams of mobile sensing agents," *Autonomous Agents and Multi-Agent Systems*, vol. 29, no. 3, pp. 495–536, 2015.

[14] T. Matsui, H. Matsuo, and A. Iwata, "Efficient methods for asynchronous distributed constraint optimization algorithm," in *Proceedings of AIA*, 2005, pp. 727–732.

[15] W. Yeoh, P. Varakantham, and S. Koenig, "Caching schemes for DCOP search algorithms," in *Proceedings of AAMAS*, 2009, pp. 609–616.

[16] P. Gutierrez and P. Meseguer, "Saving redundant messages in BnB-ADOPT," in *Proceedings of AAAI*, 2010, pp. 1259–1260.

[17] ——, "Improving BnB-ADOPT$^+$-AC," in *Proceedings of AAMAS*, 2012, pp. 273–280.

[18] ——, "BnB-ADOPT$^+$ with several soft arc consistency levels," in *Proceedings of ECAI*, 2010, pp. 67–72.

[19] P. Gutierrez, J. Lee, K. M. Lei, T. Mak, and P. Meseguer, "Maintaining soft arc consistencies in BnB-ADOPT$^+$ during search," in *Proceedings of CP*, 2013, pp. 365–380.

[20] P. Gutierrez, P. Meseguer, and W. Yeoh, "Generalizing ADOPT and BnB-ADOPT," in *Proceedings of IJCAI*, 2011, pp. 554–559.

[21] S. Arnborg, D. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM Journal of Discrete Mathematics*, vol. 8, no. 2, pp. 277–284, 1987.

[22] Y. Hamadi, C. Bessière, and J. Quinqueton, "Distributed intelligent backtracking," in *Proceedings of ECAI*, 1998, pp. 219–223.

[23] R. Wallace and E. Freuder, "Stable solutions for dynamic constraint satisfaction problems," in *Proceedings of CP*, 1998, pp. 447–461.

[24] A. Holland and B. O'Sullivan, "Weighted super solutions for constraint programs," in *Proceedings of AAAI*, 2005, pp. 378–383.

[25] R. Mailler, "Comparing two approaches to dynamic, distributed constraint satisfaction," in *Proceedings of AAMAS*, 2005, pp. 1049–1056.

[26] G. Billiau and A. Ghose, "SBDO: A new robust approach to dynamic distributed constraint optimisation," in *Proceedings of PRIMA*, 2009, pp. 641–648.

[27] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan, "Comparing performance of distributed constraints processing algorithms," in *Proceedings of the Distributed Constraint Reasoning Workshop*, 2002, pp. 86–93.

[28] A. Chechetka and K. Sycara, "An any-space algorithm for distributed constraint optimization," in *Proceedings of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, 2006, pp. 33–40.