

Solving Distributed Constraint Optimization Problems Using Cooperative Mediation*

Roger Mailler and Victor Lesser
University of Massachusetts
Department of Computer Science
Amherst, MA 01003
{mailler, lesser}@cs.umass.edu

Abstract

Distributed Constraint Optimization Problems (DCOP) have, for a long time, been considered an important research area for multi-agent systems because a vast number of real-world situations can be modeled by them. The goal of many of the researchers interested in DCOP has been to find ways to solve them efficiently using fully distributed algorithms which are often based on existing centralized techniques. In this paper, we present an optimal, distributed algorithm called optimal asynchronous partial overlay (OptAPO) for solving DCOPs that is based on a partial centralization technique called cooperative mediation. The key ideas used by this algorithm are that agents, when acting as a mediator, centralize relevant portions of the DCOP, that these centralized subproblems overlap, and that agents increase the size of their subproblems as the problem solving unfolds. We present empirical evidence that shows that OptAPO performs better than other known, optimal DCOP techniques.

1. Introduction

For a number of years now, researchers in distributed problem solving have struggled with the question of how to find an optimal assignment to a set of variables spread over a number of agents which have interdependencies. Out of this work, a number of formulations have arisen including the distributed partial constraint satisfaction problem (DPCSP)[2], distributed valued constraint sat-

isfaction problem [4], and the distributed constraint optimization problem (DCOP) [8].

A number of powerful, distributed algorithms have been developed that solve these problems either optimally, or close to optimally. For example, Distributed Depth-first Branch and Bound (DDBB) and Distributed Iterative Deepening (DID) [8], Synchronous Branch and Bound (SBB) and Iterative Distributed Breakout (IDB) [2], and the Asynchronous Distributed Optimization (Adopt) algorithm [6]. Each of these algorithms has, at their core, two common threads. First, their basic design originated directly from an associated centralized algorithm and second, they maintain total separation of the agents' knowledge during the problem solving process.

In this paper, we present a cooperative, mediation-based DCOP protocol, called Optimal Asynchronous Partial Overlay (OptAPO), that allows the agents to extend and overlap the context that they use for making their local decisions as the problem solving unfolds. When an agent acts as a mediator, it computes a solution to a portion of the overall problem and recommends value changes to the agents involved in the *mediation session*. This algorithm, like its DCSP variant APO [5], provides rapid, distributed, asynchronous problem solving without the explosive communications overhead normally associated with current distributed algorithms. OptAPO represents a new methodology that simultaneously exploits the speed of centralized techniques and the ability of distributed problem solving to identify problem substructure. In the graph coloring domain, this algorithm performs better, both in terms of communication and computation, than the Adopt algorithm which is currently the fastest known complete DCOP technique.

In the rest of this paper, we present a formalization of the DCOP problem. We then present the OptAPO algorithm and present an example of its execution on a simple problem. Next, we present the results of extensive testing that compares OptAPO with Adopt within the commonly used graph coloring domain. Lastly, we discuss some of

* Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-2-0525 and F30602-03-C-0010. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright annotation thereon.

```

procedure initialize
   $d_i \leftarrow \text{random } d \in D_i$ ;
   $F_i^* \leftarrow 0$ ;
   $p_i \leftarrow \text{sizeof}(\text{neighbors})$ ;
   $m_i \leftarrow \text{active}$ 
   $mediate \leftarrow \text{none}$ ;
  add  $x_i$  to the good_list;
  send (init,  $(x_i, p_i, d_i, m_i, D_i, C_i, \text{path}_{i,j})$ ) to neighbors;
   $\text{initList} \leftarrow \text{neighbors}$ ;
end initialize;

when received (init,  $(x_j, p_j, d_j, m_j, D_j, C_j, [\text{path}_{j,i}])$ ) do
  Add  $(x_j, p_j, d_j, m_j, D_j, C_j, \text{path}_{j,i})$  to agent_view;
  if  $x_j$  is a neighbor of some  $x_k \in \text{good\_list}$  do
    add  $x_j$  to the good_list;
    add all  $x_l \in \text{agent\_view}$  and  $x_l \notin \text{good\_list}$ 
    that can now be connected to the good_list;
     $p_i \leftarrow \text{sizeof}(\text{good\_list})$ ;
  end if;
  if  $x_j \notin \text{initList}$  do
    send (init,  $(x_i, p_i, d_i, m_i, D_i, C_i)$ ) to  $x_j$ ;
  else
    remove  $x_j$  from initList;
  check\_agen\_tview;
end do;

when received (value?,  $(x_j, p_j, d_j, m_j, c_j)$ ) do
  update agent_view with  $(x_j, p_j, d_j, m_j, c_j)$ ;
  check\_agen\_tview;
end do;

```

Figure 1. The OptAPO procedures for initialization, receiving “init” messages, and receiving “value?” messages.

our conclusions.

2. Distributed Constraint Optimization

A Constraint Optimization Problem (COP) consists of the following:

- a set of n variables $V = \{x_1, \dots, x_n\}$.
- discrete, finite domains for each of the variables $D = \{D_1, \dots, D_n\}$.
- a set of cost functions $f = \{f_1, \dots, f_m\}$ where each $f_i(d_{i,1}, \dots, d_{i,j})$ is function $f_i : D_{i,1} \times \dots \times D_{i,j} \rightarrow \mathbb{N} \cup \infty$.

The problem is to find an assignment $A^* = \{d_1, \dots, d_n | d_i \in D_i\}$ such that the global cost, called F , is minimized. Although the algorithm presented will work for any associative, commutative, monotonic aggregation function defined over a totally ordered set of values, with min and max elements, in this paper, F is defined as follows

$$F(A) = \sum_{i=1}^m f_i(A)$$

In the distributed version of this problem, DCOP, each agent is assigned one or more variables along with the functions on their variables. Overall, COP and DCOP have both been shown to be NP-complete, making search

```

procedure check\_agent\_view
  if  $\text{initList} \neq \emptyset$  or  $mediate \neq \text{none}$  do
    return;
   $c'_i \leftarrow \{x_j | \exists f_i x_j \in f_i \wedge f_i > f_i^*\}$ ;
   $m'_i \leftarrow \text{none}$ ;
  //Compute my new mediate intentions
  if  $(F_i > F_i^* \text{ and } \exists_j (f_j < f_j^* \wedge p_j \leq p_i))$  do
     $m'_i \leftarrow \text{active}$ 
  else if  $F_i > F_i^*$  do
     $m'_i \leftarrow \text{passiv e}$ 

  //Active mediate and I'm not expecting from a higher priority
  if  $m'_i = \text{active}$  and  $\neg \exists_j (p_j > p_i \wedge m_j = \text{active})$ 
    if  $\exists (d'_i \in D_i) (d'_i \cup \text{agent\_view} \text{ causes } F_i = F_i^*)$ 
      and changes are with lower priority neighbors
       $d_i \leftarrow d'_i$ ;
       $m_i \leftarrow \text{none}$ ;
       $c'_i \leftarrow \{x_j | \exists f_i x_j \in f_i \wedge f_i > f_i^*\}$ ;
      send (value?,  $(x_i, p_i, d_i, m_i, c'_i)$ ) to all  $x_j \in \text{agent\_view}$ ;
    else
      do mediate( $m'_i$ );

  //Passive mediate
  else if  $m'_i = \text{passiv e}$ 
    do mediate( $m'_i$ );

  //Mediate flag or conflict set change
  else if  $m_i \neq m'_i$  or  $(m_i = \text{none} \text{ and } c_i \neq c'_i)$ 
     $m_i \leftarrow m'_i$ ;
    send (value?,  $(x_i, p_i, d_i, m_i, c'_i)$ ) to all  $x_j \in \text{agent\_view}$ ;

  //Nothing to do, see if I need to update my good_list
  else if  $m_i = \text{none}$ 
    for  $\forall x_j, x_k x_j \in \text{agent\_view} \wedge x_k \in c_j \wedge x_k \notin \text{good\_list} \wedge m_j = \text{none}$  do
      for  $\forall x_l$  on the path to  $x_k \wedge x_l \notin \text{agent\_view}$  do
        send (init,  $(x_i, p_i, d_i, m_i, D_i, C_i, \text{path}_{i,l})$ ) to  $x_l$ ;
        add  $x_l$  to initList;
      end do;
    end do;
     $c_i \leftarrow c'_i$ ;
  end check\_agent\_view;

```

Figure 2. The OptAPO check_agent_view procedure.

In this paper, we restrict ourselves to the case where each agent is assigned a single variable and is given knowledge of its functional relationship with other neighboring variables. Since each agent is assigned a single variable, we will refer to the agent by the name of the variable it manages. Also, we restrict ourselves to considering only binary functions which are of the form $f_i(x_{i1}, x_{i2})$. Our approach can be extended to handle cases where one or both of these restrictions are removed.

Throughout the paper, we use the term *relationship graph* to refer to the graph formed by representing the agents as nodes and the functional relationships as edges. The term *neighbors* is used to refer to agents that are connected by an edge in the relationship graph.

3. Optimal APO

Figures 1, 2, 3, 4, and 5 present the OptAPO algorithm. OptAPO works by constructing a *good_list* and maintain-

```

procedure mediate( $m_i^l$ )
  preferences  $\leftarrow \emptyset$ ;
  counter  $\leftarrow 0$ ;
  for eac  $h x_j \in \text{good\_list}$  do
    send (evaluate?, ( $x_i, p_i, m_i^l$ )) to  $x_j$ ;
    counter ++;
  end do;
  mediate  $\leftarrow m_i^l$ ;
   $m_i \leftarrow m_i^l$ ;
end mediate;

when receive (w ait!, ( $x_j, p_j$ )) do
  counter --;
  if counter == 0 do c hoosesolution;
end do;

when receive (evaluate!, ( $x_j, p_j$ , labeled  $D_j$ )) do
  record ( $x_j$ , labeled  $D_j$ ) in preferences;
  counter --;
  if counter == 0 do c hoosesolution;
end do;

```

Figure 3. The procedures for mediating in OptAPO.

ing a structure called the *agent_view*. The *agent_view* stores the names, values, domains and functional relationships of agents in the environment that are linked to the owner of the *agent_view*. The *good_list* holds the names of the agents that the owner has identified either a direct or indirect relationship to through one or more functional relations in the relationship graph.

As the problem solving unfolds, each agent tries to improve the value of the subproblem they have centralized within their *good_list* or to justify its cost by identifying over-constrained structures in the relationship graph. To do this, agents take the role of the mediator, compute the optimal value of their subsystem, and attempt to change the assignments of the variables within the session to achieve this optimal value. Whenever, this cannot be achieved without causing cost for agents outside of the session, the mediator links with those agents assuming that they are involved in related cost-justifying substructure. This process continues until each of the agents have justified the cost of their centralized subproblem and they have ensured that this centralized portion contains all of the cost-barring substructures that they are part of.

In order to facilitate the problem solving process, each agent has a dynamic priority that is based on the size of their *good_list* (ties are broken by the ordering of the agents names). Priorities are used by the agents to decide the order in which one or more agents mediate when they have a need. Priority ordering is important for two reasons. First, priorities ensure that the agents with the most knowledge get to make the decisions. This improves the efficiency of the algorithm by decreasing the effects of myopic decision making. Second, priorities improve the effectiveness of the mediation process. Because lower priority agents expect higher priority agents to mediate, they are less likely to be involved in a session when the mediation request is sent.

```

procedure choose_solution
  select a solution  $s$  using a Branch and Bound search that:
  1. minimizes the cost for the agents in the good_list
  2. minimizes the cost for the agents not in the session;
   $F_s^* \leftarrow \text{cost}(s)$ ;
   $F_i^h \leftarrow F_i + \text{current cost for agents not in the session}$ ;
   $F_s^l \leftarrow F_s^* + \text{cost of using } s \text{ for agents not in the session}$ ;
  if mediate == active and  $F_s^l \leq F_i^l$  do
     $d_i \leftarrow d_i^l$ ;
  for eac  $h x_j \in \text{agent\_view}$  do
    if  $x_j \in \text{preferences}$  do
      if  $d_j^l \in s$  violates an  $x_k$  and  $x_k \notin \text{agent\_view}$  do
        send (init!, ( $x_i, p_i, d_i, m_i, D_i, C_i, \text{path}_{i,k}$ )) to  $x_k$ ;
        add  $x_k$  to initList;
      end if;
      if mediate == active and  $F_s^l \leq F_i^l$  do
        send (accept!, ( $d_j^l, x_i, p_i, d_i$ )) to  $x_j$ ;
        update agent_view for  $x_j$ ;
      else if mediate == active and  $F_s^l > F_i^l$  do
        send (accept!, ( $d_j, x_i, p_i, d_i$ )) to  $x_j$ ;
      end if;
    else if mediate == active do
      send (value?, ( $x_i, p_i, d_i, m_i, c_i$ )) to  $x_j$ ;
    end if;
  end do;
  mediate  $\leftarrow$  none;
  check agent_view;
end choose_solution;

```

Figure 4. The procedure for choosing a solution.

3.1. Initialization (Figure 1)

On startup, the agents are provided with the value (they pick it randomly if one isn't assigned) and the functions on their variable. Initialization proceeds by having each of the agents, i , send out an "init" message to each of its neighbors, j . This initialization message includes the variable's name (x_i), priority (p_i), current value (d_i), domain (D_i), and functional relationships (C_i). Also included in this message are the variable m_i , which indicates the agents current desire to mediate, and a list of agents that lie between i and j in the relationship graph. The purpose of both of these pieces of information will be described below. The array *initList* records the names of the agents that initialization messages have been sent to, the reason for which will become immediately apparent.

When an agent receives an initialization message (either during the initialization or through a later link request), it records the information in its *agent_view* and adds the variable to the *good_list* if it can. An agent is only added to the *good_list* if it is connected to another agent already in the list through a functional relationship. This ensures that the graph created by the agents in the *good_list* always remains connected. The *initList* is then checked to see if this message is a link request or a response to a link request. If the agent is not in the *initList* then it means this is a link request, so a response "init" is generated and sent. If an agent is in the *initList*, it means that this message is a response to a request that was previously sent. In this case, a response message is not generated. This mechanism prevents the agents from sending

```

when received (evaluate?,  $(x_j, p_j, m_j)$ ) do
  update agent_view with  $(x_j, p_j, m_j)$ ;
  if (mediate  $\neq$  none or  $\exists_k (p_k > p_j \wedge m_k == \text{active})$ )
    and  $m_j == \text{active do}$ 
      send (wait!  $(x_i, p_i)$ );
    else
      if mediate  $\neq$  active do
        mediate  $\leftarrow m_j$ ;
        label each  $d \in D_i$  with the names of the agents
          and associated costs incurred by setting  $d_i \leftarrow d$ ;
        send (evaluate!  $(x_i, p_i, \text{labeled } D_i)$ );
      end if;
    end do;

when received (accept!,  $(d, x_j, p_j, d_j, m_j)$ ) do
   $d_i \leftarrow d$ ;
  mediate  $\leftarrow$  false;
  send (value?  $(x_i, p_i, d_i, m_i, c_i)$ ) to all  $x_j$  in agent_view;
  update agent_view with  $(x_j, p_j, d_j, m_j)$ ;
  check_agent_view;
end do;

```

Figure 5. Procedures for receiving a session.

“init” messages to one another in an infinite loop

At times, the agents in the *good_list* are a subset of the agents contained in the *agent_view*. This is done to maintain the integrity of the *good_list* and allow links to be bidirectional. To understand this point, consider the case when a single agent has repeatedly mediated and has extended its local subproblem down a long path in the relationship graph. As it does so, it links with agents that may have a very limited view and therefore are unaware of their indirect connection to the mediator. In order for the link to be bidirectional, the receiver of the link request has to store the name of the requester, but cannot add them to their *good_list* until a path can be identified.

In order to ensure the optimality of the algorithm, each of the agents does not terminate until all of the functional relations $f_j > f_j^*$ that include an agent within its *agent_view* appear in its *good_list*. During periods of inactivity agents try to achieve this condition by growing their *good_list* to include these “non-optimal” relations. This is how agents prevent unnecessary centralization and where the path information provided as part of the initialization message comes into play. This process ensures that the following property is enforced at the termination of the algorithm:

Property 1 $\forall x_j \forall f_j ((x_j \in \text{agent_view}_i \wedge x_j \in f_j \wedge f_j > f_j^*) \rightarrow f_j \in \text{good_list}_i)$ upon termination.

3.2. Checking the agent view (Figure 2)

After the agents receive all of the initialization messages, they execute the *check_agent_view* procedure. In this procedure, the *agent_view* (assigned, known variable values) is used to calculate the current cost, F_i , of the relationship subgraph formed by the agents within the *good_list*. If, during this check, an agent finds that F_i is greater than the optimal value of the subsystem, called

F_i^* , it conducts either an *active* or *passive* mediation session. This check ensures that the following property is obtained at the termination of the algorithm:

Property 2 $\forall_i F_i^* == F_i$ upon termination

At startup, the value of F_i^* is always initialized to 0 meaning that the best answer obtained thus far has no cost. This value changes as the agent centralizes substructures which must have some cost in the optimal answer. Agents can determine this cost because of the centralized solver used in the mediation process. Whenever an agent mediates, it recomputes the value of F_i^* by running a centralized search on its entire *good_list*.

Agents decide between an active or passive mediation based on whether or not one of the suboptimal functional relations in their *good_list* has an agent in it that is lower or equal (itself) priority to itself. If one of the agents is lower or equal priority, the mediation will be active, otherwise, it will be passive.

There are two main differences between active and passive mediation. First, during an active mediation, the receiving agent sets its *mediate* flag. This flag prevents it from starting or engaging in another active mediation until it is cleared. This causes a region of stability in the agent system which allows the mediation session to have the full effect but, reduces parallelism because it prevents other agents from mediating. The second difference is really based on the intent. The intent of passive mediation is to verify and understand the results that higher priority agents have obtained without interfering in their actions or changing their current solution. In other words, the intent of passive mediation is to change the value of F_i^* , the intent of active mediation is to change F_i^* and F_i . Passive mediation both increases the parallelism of the problem solving and allows agents to gain additional context (extend their views) without causing system instability.

If an agent decides that it wishes to actively mediate, it can only do so when it has not been told by a higher priority agent that they want to actively mediate. Agents within the system are able to tell when a higher priority agent wants to mediate because of the m_i flag mentioned in the previous section. Whenever an agent checks its *agent_view* it recomputes the value of this flag which indicates its desire to mediate in the future if given the opportunity. This information is shared with each of the agents in its *agent_view* whenever the value changes. The overall effect of the m_i flag is similar to the two-phase commit commonly seen in database systems and ensures that the protocol remains live-lock and deadlock free.

As an active mediator, an agent first attempts to rectify the suboptimal condition by changing its own variable. This simple technique prevents sessions from occurring unnecessarily, which stabilizes the system and saves messages and time. If the mediator finds a value that makes $F_i == F_i^*$ and it finds that the functional rela-

tionships being improved by the change are shared with only lower priority agents, it makes the change and sends out a “value?” message to the agents in its *agent_view*. If cannot find such a value, it starts an active mediation session which is described in section 3.3.

A “value?” message is similar to an “init” message, in that it contains information about the priority, current value, etc. of a variable. Unlike the “init” message however, “value?” messages contain a list, c_i , which gives the name of any agent which shares a relation with the sender that has a cost greater than 0. Using this list, an agent can tell when conflicts exist that may need to be included in their *good_list*. This allows the agents to prevent unnecessary centralization when the situation clearly indicates that it has no direct benefit. In other words, there is no need to add an agent to the local subsystem when they are not involved in a relationship which is increasing the global cost.

3.3. Mediation (Figures 3, 4, and 5)

The most complex and certainly most interesting part of the protocol is the mediation session. As was previously mentioned in this section, an agent decides to mediate if it finds that $F_i > F_i^*$. The mediation session starts with the mediator sending out “evaluate?” messages to each of the agents in its *good_list*. The purpose of this message is two-fold. First, it informs the receiving agent that a mediation is about to begin and, as mentioned earlier, if the mediation is active, tries to obtain a lock from that agent. The second purpose of the message is to obtain information from the agent about the effects of making them change their local value. This is a key point. By obtaining this information, the mediator gains information about variables and relationships outside of its local view without having to directly and immediately link with those agents.

When an agent receives a mediation request, it will respond with either a “wait!” or “evaluate!” message. Agents respond with a “wait” message whenever the request is for an active session and the agent is either currently involved in another active session or is expecting a request for an active session from an agent that is higher priority than the requester. This allows for a great deal of parallelism because all passive requests are responded to and active requests are only turned away when absolutely needed. Whenever it can, the agent evaluates each of its domain elements and labels them with the names of the agents that would have a shared functional relation with cost $f_i > f_i^*$ along with that cost if it were asked to take that value. In the graph coloring domain, the labeled domain can never exceed $O(|D_i| + n)$. This information is returned in an “evaluate!” message. It should be noted that, although in this implementation, the agents need not return all of the names if for security reasons they wish not to. This affects the optimality of the al-

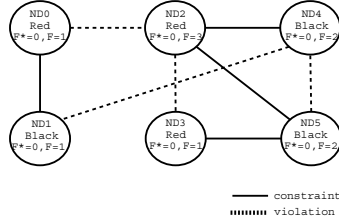
gorithm because it prevents agents from gaining enough context, but does provide some degree of autonomy and privacy to the agents.

When the mediator has received a response from all of the agents that it has sent a request to (the *counter* variable reaches 0), it chooses a solution. Agents that sent a “wait!” message are dropped from the mediation, but the mediator attempts to fix whatever problems it can based on the information it receives from the remaining agents. The mediator then conducts a Branch and Bound search [1] which, as a primary criteria, minimizes the cost of the subproblem in the *good_list* and as a secondary criteria minimizes the costs for agents outside of the session. The results of minimizing the primary criteria, being the optimal value for the subproblem in the *good_list*, is saved as F_i^* .

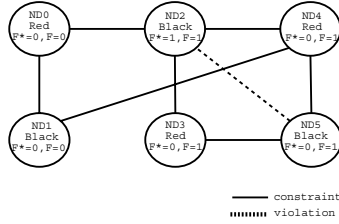
The agents employ two special tactics during this search. First, the values are ordered so that the first branch of the search is the current solution. This usually causes the bound to become very close to F_i^* after it is explored taking advantage of previous work that has been done on the problem. This has the overall effect of improving the search speed as was reported in [7]. The second tactic terminates the search early whenever the bound is equal to the current F_i^* and the cost for agents outside of the mediation is 0. This method works because the current F_i^* is always an lower bound on the actual F^* and F_i^* only increases on successive search because of the monotonic nature of the number of variables in the search and the aggregation function. This method considerably reduces the effort used during this search.

At the conclusion of the local search, the mediator computes two values, F_i' and F_s' . The value F_i' is the cost, given the current set of variable values, for the extended subsystem which is composed of the agents within the *good_list* along with any other agents which appear in the preference information returned in the “evaluate!” messages. The value F_s' is the cost for this same extended subsystem given the solution s returned by the centralized search. Ideally, these values will be the same. In other words, s has a non-negative effect on the current state of the global problem. Because agents act myopically when they compute s , they sometime choose a solution which appears to be good, but overall has a very negative effect. Whenever this happens, the mediator ignores s (keeping any changes to F_i^* however) and reverts to the current solution, effectively preventing itself from making a locally optimal decision that has obviously bad global consequences. The overall effect is similar to passive mediation in that it maintains stability in the system while the agent gains context information.

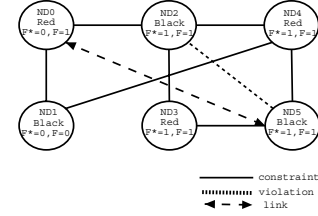
After computing these values, the mediator links (sends “init” messages) with any agent that is not in its *agent_view* and has been forced to have an increased cost as result of s . This occurs even if the medi-



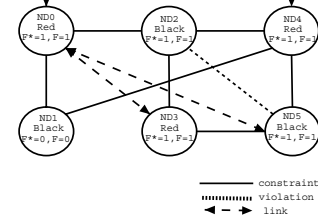
(a) Startup



(c) After ND2 Mediates



(b) After ND5 Mediates



(d) Termination

ator c chooses not to use. This step expands the agent's *good_list* so that the next time it mediates, it does not repeat the same mistake. The mediator concludes the session by sending “accept!” messages (if the session was active) to the agents in the session, who, in turn, adopt the proposed answer and “value?” messages to all of the other agents in its *agent_view*.

3.4. An Example

Consider the 2-coloring problem in figure 6(a) which was chosen to illustrate the algorithm's features without being overly complicated. In this problem there are 6 variables, each assigned to a different agent, and 8 functional relations between the variables. Each of the functional relations is a ‘not equals’ and has a cost of 1 for being violated and 0 otherwise. The goal of the agents is to minimize the global cost F^* (the sum of the values returned by the functional relations), which in this case is 1. In other words, this is a MaxSAT problem instance.

On startup, the problem is in the state in figure 6(a). The current cost of the system is 4, because 4 of the ‘not equals’ relationships are violated (indicated by the dotted lines). Each of the agents has an internal optimal subsystem value, $F_i^* = 0$, since none of them have mediated and therefore computed the actual value.

Following the protocol, the agents send out “init” messages to each of their neighbors. So, for example, ND0 send out “init” messages to ND1, ND2, and ND3. As the “init” messages are received, the agents add each of their neighbors to their *good_list* because they have a direct path through a shared relation.

Once all of the “init” messages are received, the agents check their *agent_view*. All of the agents are able to detect the non-optimal state of the problem by computing the value of F_i . For example, ND2 computes an $F_i = 3$

because it has ND0, ND2, ND3, ND4, and ND5 in its *good_list* and can see the cost of the relations between ND0 and ND2, ND2 and ND3, as well as ND4 and ND5. This causes ND2 to set its m_i flag to **active**. In this example, each of the agents is involved in a suboptimal relation so they all set their m_i flag to **active**. ND2 is the highest priority agent in the system, it has a priority of 5, elects to take over as the mediator and begins an active session with ND0, ND3, ND4, and ND5.

As the mediator, ND2 first checks to see if it can correct the suboptimality by making a local change, which it cannot. It sends “evaluate?” messages to ND0, ND3, ND4, and ND5. Each of these agents, upon receiving the message, checks to see if it is expecting a mediation from a higher priority agent, which they are not, and then sets its *mediate* flag to **active**. They label each of their domain elements and reply with the following information each using an “evaluate!” message (since the costs are all 1, they are left out for clarity):

- ND0 - Black conflicts with ND1; Red conflicts with ND2
- ND3 - Black conflicts with ND5; Red conflicts with ND2
- ND4 - Black conflicts with ND1 and ND5; Red conflicts with ND2
- ND5 - Black conflicts with ND4; Red conflicts with ND2 and ND3

ND2 conducts a Branch and Bound search and finds that $F_i^* = 1$ for its *good_list* as well as finding a solution which has 0 conflicts external to the mediation. It tells ND4 to change its value to Red and changes its own color to Black, leaving the system in the state in figure 6(c).

All of the agents check their *agent_view* again. Several agents detect a suboptimal subsystem. In fact, ND3, ND4 and ND5 have suboptimal subsystems. ND5 set its m_i flag to **active** and ND3 and ND4 sets their m_i to **passive**, because the only cost they see is between ND2 ($p_i = 5$) and

ND5 ($p_i = 4$, tie broken on name) who are both higher priority.

This time ND5 actively mediates. ND5 is unable to locally correct the difference in its F_i and F_i^* so it sends “evaluate?” messages to ND2, ND3, and ND4. It receives the following information from those agents in the returned “evaluate!” messages:

- ND2 - Black conflicts with ND5; Red conflicts with ND0, ND3, and ND4
- ND3 - Black conflicts with ND2 and ND5; Red causes no conflicts
- ND4 - Black conflicts with ND1, ND2, and ND5; Red causes no conflicts

At the same time, ND4 starts a passive mediation session with ND1, ND2, and ND5. and ND3 starts a passive session with ND2 and ND5.

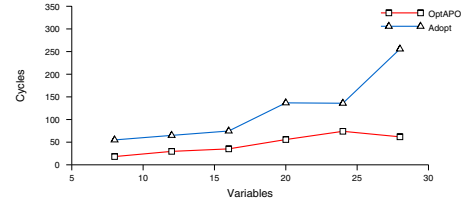
ND3, ND4, and ND5 conduct an internal searches and each find a solution with an $F_i^* = 1$, and no external conflicts. The solution they find, in fact, is identical to the current assignment which was the first branch in their search tree. ND5 sends out “accept!” messages and, since they are passively mediating, ND3 and ND4 change their internal lower bound and send “value?” messages to their neighbors

Finally, ND0 sees the conflict between ND2 and ND5. It sees that their mediate flags are set to **none**, so it links with ND5. This leaves the problem in the state in figure 6(b).

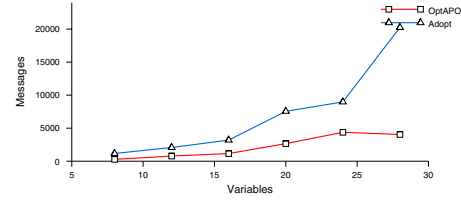
After two additional passive mediation sessions by ND0 and some additional linking, the algorithm terminates in the state in figure 6(d). Each of the agents has a $F_i^* = F_i$. Notice that only one of the agents achieves complete centralization. In this example, ND0 achieved this linking near the very end of the execution by identifying the cost between ND2 and ND5, causing it to link with ND5. This forced it to justify that cost, so it passively mediated and discovered ND3 and ND4. This caused it to link with them, allowing it to justify the total cost of 1.

4. Evaluation

To test OptAPO, we implemented the protocol in a distributed simulation environment called the Farm [3] and conducted a number of experiments in the distributed MaxSAT 3-coloring domain. We downloaded the Adopt simulator and algorithm which was constructed by Pragnesh Jay Modi, the designer of the algorithm, and verified that both of the simulators use a comparable notion of a *cycle*. In both simulators, during a cycle, all of the incoming messages are delivered and processed, and outgoing messages are queued for delivery at the beginning of the next cycle. We then instrumented the Adopt simulator to compute the number of cycles, serial runtime, and the number of messages being transmitted.



(a) Cycles



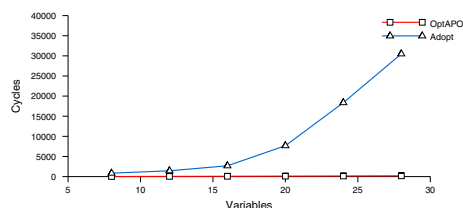
(b) Messages

Figure 7. Comparison of OptAPO and Adopt for various graph sizes at $m = 2.0n$.

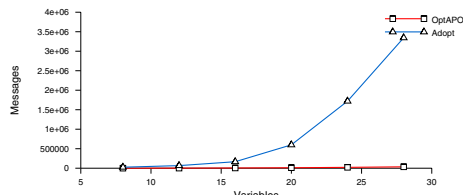
We ran two series of tests within the MaxSAT domain to compare these algorithms. In the first series of tests, we compared the number of cycles and messages used by OptAPO and Adopt. For this series, we created random graph coloring instances with $m = 2.0n$ (under-constrained) and $m = 3.0n$ (very over-constrained). We generated 100 instances at $n = 8, 12, 16, 20, 24$, and 28 for a total of 1200 individual graphs. For fairness, we used the same graphs to test both algorithms. The results of this series can be seen in figures 7(a) 7(b), 8(a), and 8(b).

In the second set of tests, we compared the actual run times of the two algorithms against each other. Both algorithms were run on a single, dedicated 2.8 GHz Pentium 4 with 512MB of RAM on a 25 instance subset of the graphs from the first series of tests. To show that OptAPO’s performance was not simply an artifact of the Branch and Bound search internal to the agents, we ran that algorithm on the graph instances as well. The results from this test can be seen in figures 9(a) and 9(b) which are logarithm scale.

As you can see from the figure, OptAPO outperforms Adopt in terms of cycles, messages, and runtime on both under and over-constrained problems. OptAPO’s superior performance can be most directly attributed to its combined use of centralization and distribution. Adopt works by iteratively trying different combinations of variables while simultaneously refining its upper and lower bound. OptAPO avoids this iterative discovery process which allows the agents to find the optimal or near optimal assignment quickly and then to verify its correctness and



(a) Cycles



(b) Messages

Figure 8. Comparison of OptAPO and Adopt for various graph sizes at $m = 3.0n$

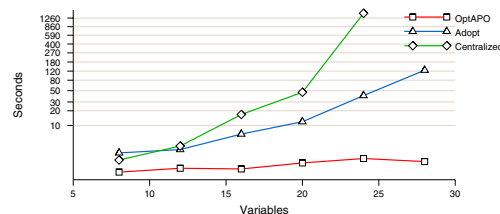
make repairs when appropriate. What you should also notice is that OptAPO actually runs faster than the Branch and Bound search that it uses internally. This is by no means meant to show that a distributed algorithm operates faster than a centralized one, but shows that the runtime characteristics of the algorithm are not simply a byproduct of the centralized search. In fact, the improvements in search time over the centralized search are most likely caused by the combination of the value ordering heuristic and early search termination method talked about in section 3.3.

5. Conclusions

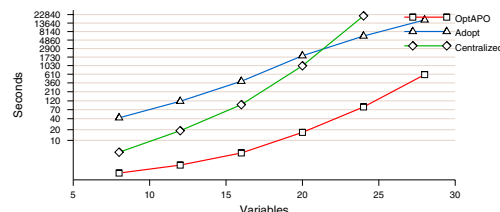
In this paper, we presented a new method for solving DCOPs called *Optimal Asynchronous Partial Overlay* (OptAPO). The key features of this technique are that agents mediate over conflicts, the context they use to make local decisions overlaps with that of other agents, and as the problem solving unfolds, the agents gain more context information along the critical paths within the problem to improve their decisions. We have shown that the OptAPO algorithm is both optimal and complete and that it performs better than the Adopt algorithm on MaxSAT graph coloring problems.

References

- [1] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21-70, 1992.
- [2] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In G. Smolka, editor, *Principles*



(a) $m = 2.0n$



(b) $m = 3.0n$

Figure 9. Comparison of runtime of OptAPO, Adopt, and centralized Branch and Bound.

and Practice of Constraint Programming (CP-97), volume 1330 of *Lecture Notes in Computer Science*, pages 222-236. Springer-Verlag, 1997.

- [3] B. Horling, R. Mailler, and V. Lesser. Farm: A Scalable Environment for Multi-Agent Development and Evaluation. *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003)*, pages 171-177, May 2003.
- [4] M. Lemaitre and G. Verfaillie. An incomplete method for solving distributed valued constraint satisfaction problems. In *Proceedings of the AAAI Workshop on Constraints and Agents 1997*.
- [5] R. Mailler and V. Lesser. A Mediation Based Protocol for Distributed Constraint Satisfaction. *The Fourth International Workshop on Distributed Constraint Reasoning*, pages 49-58, August 2003.
- [6] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proceedings of the Second International Joint Conference on Autonomous Agent and Multiagent Systems (AAMAS-03)*, Melbourne, Australia, July 2003.
- [7] R. Wallace. Enhancements of branch and bound methods for the maximal constraint satisfaction problem. In M. Jampe, E. Freduer, and M. Maher, editors, *OCS'95: Workshop on Over-Constrained Systems at CP'95* Cassis, Marseilles, 18 1995.
- [8] M. Yokoo and E. H. Durfee. Distributed constraint optimization as a formal model of partially adversarial cooperation. Technical Report CSE-TR-101-91, University of Michigan, Ann Arbor, MI 48109, 1991.