A Mediation Based Protocol for Distributed Constraint Satisfaction *

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

Abstract

Distributed Constraint Satisfaction Problems (DisCSP) have long been considered an important area of research for multi-agent systems. This is partly due to the fact that many real-world problems can be represented as a constraint satisfaction problem and partly because real-world problems often present themselves in a distributed form. The agent paradigm is particularly well suited to handling problems of this type. Agents are easily distributable and have both encapsulated state as well as autonomous reasoning capabilities. This allows them to work together to solve problems that cannot be completely centralized due to security, dynamics, or complexity. In this paper, we present an algorithm called *asynchronous partial overlay (APO)* for solving DisCSPs that is based on a mediated negotiation process. The primary ideas behind this algorithm are that agents, when acting as a local mediator, partially centralize a subproblem of the CSP, that these partially centralized subproblems overlap, and that agents increase the size of their subproblems along critical paths within the CSP as the problem solving unfolds.

1 Introduction

Distributed constraint satisfaction has become a classic paradigm to describe a myriad of distributed problems including distributed resource allocation [3; 13], distributed scheduling [14], and distributed interpretation [7]. It's no wonder that a vast amount of effort and research has gone into creating algorithms, such as distributed breakout (DBO) [19], asynchronous backtracking (ABT) [18; 17], and asynchronous weak-commitment (AWC) [16], for solving these problems.

However, almost all of the previous work is DisCSPs can be placed into one of three major categories based on the type of decomposition that is being done [9]. The first major decomposition type is based on breaking apart the search space by assigning particular domain elements from one or more of the variables to individual agents. Examples of this technique can be found in [1; 4; 8]. The principle drawback of this technique is that each of the agents has to know the variables, domains, and constraints for the entire problem. The second type, known as functional decomposition, involves breaking the search apart based on the functions of the search. For example, if we were doing search while trying to maintain arc-consistency, one processor might conduct the search while others conduct the consistency checking in parallel. One example of this is presented in [9]. The last type and the one that has received the most attention is variable decomposition [3; 17; 16; 18; 19]. Variable decomposition involves assigning each agent one or more variables to manage giving each knowledge of the constraints on their variables.

A common thread to all of these techniques is that they each provide the agents with a total functional, variable, or domain decomposition thereby preventing the agents from making informed local decisions about their individual portion of the overall search. For example, in variable decomposition, agents are only aware of their own variables and the constraints on them. Agents have no basis for understanding why another agent is unable to accept the value of their variable. In this paper, we present a cooperative mediation based negotiation protocol, called Asynchronous Partial Overlay (APO), that allows the agents to extend and overlap the context that they use for making their local decisions. This variable based decomposition technique allows for rapid distributed asynchronous problem solving without the explosive communications overhead normally associated with this decomposition technique. In preliminary testing, using the graph coloring domain, this algorithm performs better, both in term of terms of communication and computation, than the AWC algorithm. This is particularly true for problems that are considered "critically" constrained.

In the rest of this paper, we present a formalization of the DisCSP problem. We then present the APO algorithm and

*The effort represented in this paper has been sponsored 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. 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 notwithstanding any copyright annotation thereon.

```
procedure initalize
  d_i ← random d ∈ D_i;
  send (init, (x_i, d_i, D_i, constraints_i)) to neighbors;
  initList ← neighbors;
end initialize;

when received (init, (x_j, d_j, D_j, constraints_j)) do
  Add (x_j, d_j, D_j, constraints_j) to agent_view;
  if x_j is a neighbor of some x_k ∈ good_list do
    Augment the good_list with (x_j, D_j);
    for all x_l that is neighbor of x_j do
      if x_l ∈ agent_view and x_l ∉ good_list do
        Augment the good_list with (x_l, D_l);
    end do;
    if good_list == ∅ do
      broadcast no solution;
  end if;
  if x_j ∉ initList do
    send (init, (x_i, d_i, D_i, constraints_i)) to x_j;
  else
    remove x_j from initList;
  check_agent_view;
end do;
```

Figure 1: The APO procedures for intialization and linking.

```
when received (ok?, (x_j, d_j, D_j)) do
  update agent_view with (x_j, d_j);
  if x_j ∈ good_list do
    update good_list with D_j;
    if good_list == ∅ do
      broadcast no solution;
  check_agent_view;
end do;

procedure check_agent_view
  if initList ≠ ∅ do
    return;
  when x_i in conflict with a neighbor
      and not expecting a negotiation (see section 3.2)
      and negotiate ≠ true do
    if ¬∃(d_i ∈ D_i) (d_i ∪ agent_view ∈ good_list)
      do negotiate;
    else select d_i ∈ D_i that does not conflict;
      send (ok?, (x_i, d_i, D_i)) to all x_j ∈ agent_view;
  end do;
end check_agent_view;
```

Figure 2: The procedures for doing local resolution, updating the *agent_view* and the *good_list*.

discuss the issues of soundness and completeness as well as presenting an example of the execution on a simple problem. Next, we present some initial tests that compare APO with AWC within the commonly used graph coloring domain. Lastly, we discuss some of our conclusions and future work on the protocol.

2 Distributed Constraint Satisfaction

A Constraint Satisfaction Problem (CSP) consists of the following:

- a set of *n* variables $V = \{x_1, \ldots, x_n\}$.
- discrete, finite domains for each of the variables $D = \{D_1, \ldots, D_n\}$.
- a set of constraints $R = \{R_1, \ldots, R_m\}$ where each $R_i(x_{i1}, \ldots, x_{ij})$ is a predicate on the Cartesian product $D_{k1} \times \cdots \times D_{kj}$ that returns true iff the value assignments of the variables satisfies the constraint.

The problem is to find an assignment $A = \{d_1, \ldots, d_n | d_i \in D_i\}$ such that each of the constraints in R is satisfied. Overall, CSP has been show to be NP-complete, making some form of search a necessity.

In the distributed case, using variable based decomposition, each agent is assigned one or more variables along with the constraints on their variables. The goal of each agent, from a local perspective, is to ensure that each of the constraints on its variables are satisfied. It should be fairly clear that for each of the agents, achieving this goal is not independent of the goals of the other agents in the system. In fact, in all but the simplest cases, the goals of the agents are strongly interrelated. For example, in order for one agent to satisfy its local constraints, another agent, potentially not directly related through a constraint, may have to change its variables value.

In this paper, for the sake of clarity, we restrict ourselves to the case where each agent is assigned a single variable and is given knowledge of the constraints on that variable. 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 constraints. It is fairly easy to extend our approach to handle the case where one or both of these restrictions are removed.

Definition 1 *A binary CSP is a CSP where all of the constraints in R are of the form* $R_i(x_{i1}, x_{i2})$.

Definition 2 *The constraint graph of a binary CSP is a graph* $G =< V, E >$ *where V is the set of variables in the CSP and E is the set of edges representing the set of constraints in R. i.e.* $R_i(x_{i1}, x_{i2}) \in R \rightarrow (x_{i1}, x_{i2}) \in E$

3 Asynchronous Partial Overlay

3.1 Basic Concepts

The key concepts behind the creation of the APO algorithm are

- Using mediation, agents can solve subproblems of the DisCSP through local search.
- These local subproblems can and should overlap to allow for more rapid convergence of the problem solving.
- Agents should, over time, increase the size of the subproblem they work on along critical paths within the CSP. This activity increases the overlap with other agents and ensures the completeness of the search.

To do these things, the agents share information about the constraints they have on their variable assignments and update the views of other agents whenever they alter their local value. In addition, each of the agents maintains a *good_list*

```
procedure negotiate
   preferences ← ∅;
   counter ← 0;
   for each x_j ∈ good_list do
      D'_j ← domain for x_j given good_list
      send (evaluate?, (x_i, D'_j)) to x_j;
      counter ++;
   end do;
   negotiate ← true;
end negotiate;

when receive (wait!, (x_j)) do
   counter --;
   if counter == 0 do choose_solution;
end do;

when receive (evaluate!, (x_j, labeled D_j)) do
   record (x_j, labeled D_j) in preferences;
   counter --;
   if counter == 0 do choose_solution;
end do;
```

Figure 3: The procedures for mediating an APO negotiation.

```
procedure choose_solution
   select s ∈ good_list s.t. the least number of agents
         outside of the negotiation will be violated
         according to preferences;
   for each x_j ∈ agent_view do
      if x_j ∈ preferences do
         if d_j ∈ s violates an x_k
               and x_k ∉ agent_view do
            send (init, (x_i, d_i, D_i, constraints_i)) to x_k;
            add x_k to initList;
         end if;
         send (accept!, (x_j, d_j ∈ s, x_i, d_i)) to x_j;
         update agent_view for x_j
      else
         send (ok?, (x_i, d_i, D_i)) to x_j;
      end if;
   end do;
   negotiate ← false;
   check_agent_view;
end choose_solution;
```

Figure 4: The procedure for choosing a solution during an APO mediation.

which contains solutions to the subproblem that the agent has gained a knowledge of. The $good\_list$ has a number of interesting properties which are easily exploitable in a cooperative setting. For example, agents can quickly identify when a problem is over-constrained whenever they create an empty $good\_list$. In addition, the $good\_list$ provide consistency checking, which is exploited as part of the mediation process by propagating the removal of domain elements to other agents.

3.2 The Algorithm

Figures 1, 2, 3, 4, and 5 present the basic APO algorithm. The algorithm works by constructing a *good_list* and maintaining a structure called the $agent\_view$. The $agent\_view$ holds the names and some information about the values, domains and constraints of agents in the environment that are linked to the owner of the $agent\_view$. The *good_list* holds the set of acceptable solutions to the subproblem that the agent has thus far identified.

Definition 3 *The good_list contains only valid solutions for the agents and their shared constraints. In addition, the constraint graph represented in the* $good\_list$ *is connected.*

In order to facilitate the problem solving process, each agent is assigned a static priority based on their name. The priority is used in deciding who mediates a negotiation when a conflict arises. This is not key to the success of the algorithm, but causes higher priority agents to mediate more often and more effectively. By having them mediate more often higher priority agents are able quickly extend their connected context. They mediate more effectively because lower priority agents expect the negotiation, decreasing the likelihood that they will respond with a "wait!" message when a mediation session is stared.

Unlike the priorities assigned to agents in ABT and AWC, priorities is this protocol are only loosely related to an ordering of the agents. It should become clear later in this section that the more often an agent has conflict, the larger the subproblem it has knowledge of and the more agents it mediates over. This is, in effect, very similar to the dynamic priorities in AWC. Priorities is this protocol are used to try to force higher priority agents to gain the context, but do not prevent lower priority agents from achieving. In the future, we plan to explore the question of dynamically changing the priorities, possibly based on the size of the subproblem an agent knows about, to further facilitate the protocols convergence.

Initialization (Figure 1)

On startup, the agents are only provided with only the value of their variable (they pick it randomly if one isn't assigned) and the constraints on their variable. Initialization proceeds by having each of the agents send out an "init" message to each of its neighbors. This initialization message includes the name of the variable, the variables current value, the domain of the variable, and the set of constraints on the variables value. 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 its $agent\_view$, adds the variable to the ever expanding *good_list*, and sends a reply initialization message to the sender if it does not appear in the $initList$ array. Adding an agent to the *good_list* entails recomputing (actually augmenting) the set of acceptable solutions given the new variable, its domain, and the constraints on its value. The $initList$ array prevents the agents from sending initialization messages to each other in an infinite loop. If the agent was in the $initList$, it is removed. Note that if the *good_list* becomes empty after adding the new variable and constraints

```
when received (evaluate?, (x_j, D'_i)) do
  D_i ← D'_i;
  if good_list == ∅ do
    broadcast no solution;
  if negotiate == true or
      x_i expecting a negotiation from a
      higher priority agent than x_j do
    send (wait!, (x_i));
  else
    negotiate ← true;
    label each d ∈ D_i with the names of the agents
      that would be violated by setting d_i ← d;
    send (evaluate!, (x_i, labeled D_i));
  end if;
end do;

when received (accept!, (x_i, d, x_j, d_j)) do
  d_i ← d;
  negotiate ← false;
  send (ok?, (x_i, d_i, D_i)) to all x_j in agent_view;
  update agent_view with (x_j, d_j);
  check_agent_view;
end do;
```

Figure 5: Procedures for receiving an APO negotiation.

to the local subproblem, sufficient information has been gathered to prove that no solution to the global problem can exist. The agent broadcasts "no solution" to its neighbors if this ever happens.

It is important to note that the agents contained 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 constraint 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 $good\_list$, without violating definition 3, until they can identify a path connecting them to the requester.

Checking the agent view (Figure 2)

After an agent receives all of the initialization messages it is expecting from its neighbors, it checks to see if any conflicts exist by executing the the check_agent_view procedure. In this procedure, the current $agent\_view$ (assigned, known variable values) is checked to make sure that no constraints are violated between the agent and its neighbors. The $agent\_view$ is also checked to find out if a higher priority agent in the $good\_list$ has a conflict. This is done because it is assumed that whenever this happens, the higher priority agent will start a session. If it is discovered that the agent has conflicts with only lower priority agents (is not expecting a negotiation), then it checks to see if it can resolve the conflicts by making a simple local change. If it can, the agent changes its value and sends out "ok?" messages. This simple, but effective check prevents a negotiation from occurring, stabilizing the overall system, saving messages and time.

An "ok?" message contains the name of a variable, its current value, and what is believed to be the current allowable variable domain. When agents receive an "ok?" message, they update their $agent\_view$ and if the agent is in the $good\_list$, update the $good\_list$ with the new domain. This seemingly unimportant step forms part of the constraint propagation mechanism that was mentioned in section 3.1. Notice that as a result of this update message the $good\_list$ can become empty.

If during the execution of the check_agent_view procedure the agent discovers it has a conflict, cannot resolve it locally, and is not expecting a negotiation, it takes over as mediator and begins the negotiation process.

Mediated negotiation (Figures 3, 4, and 5)

The most complex and certainly most interesting part of the protocol is the negotiation. As was previously mentioned in this section, an agent decides to mediate if it is in conflict with one of its neighbors and is not expecting a negotiation from a higher priority agent. The negotiation starts with the mediator sending out "evaluate?" messages to each of the agents in its $good\_list$. The purposes of this message are three-fold. First, it informs the receiving agent that a mediation is about to begin and tries to obtain a lock from that agent. This lock, referred to as $negotiate$ in the figures, prevents the agent from engaging in two negotiation session simultaneously or from doing a local value change during the course of a session. 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 constraints outside of its local view without having to directly and immediately link with those agents. The final reason is that the mediator informs the agents about their domain according to the mediator's view. This forms the second half of the constraint propagation mentioned earlier.

When an agent receives a negotiation request, it will either respond with a "wait!" message or a "evaluate!" message. The "wait" message indicates to the requester that the agent is currently involved in a negotiation session or is expecting a negotiation from a higher priority agent than the requester. If, on the other hand, the agent is available to negotiate, it labels each of the remaining domain elements with the names of the agents that the receiver would be in conflict with if it were asked to take that value.[1] This information is returned in the "evaluate!" message. It should be noted that, although in this implementation, the agents need not return all of the names if for security reasons it wishes not to. This effects the completeness of the algorithm, but does provides some degree of autonomy and privacy to the agents.[2]

When the mediator has received either a "wait!" or "evaluate!" message from all of the agents that it has sent a re-

[1] In the graph coloring domain, the labeled domain can never exceed $O(|d_i| + n)$.

[2] The completeness proof relies on an agent eventually obtaining a complete view of the problem, if this doesn't happen, the algorithm cannot be complete.

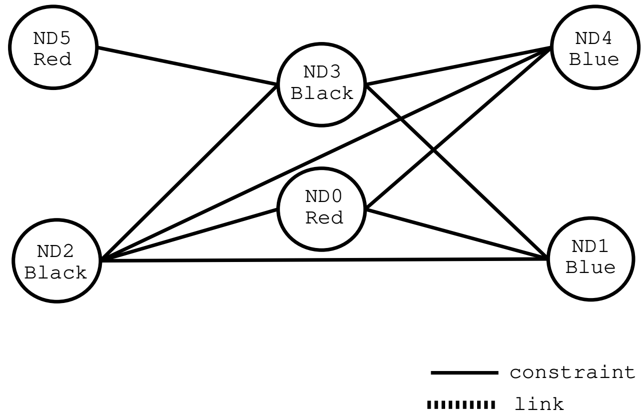

Figure 6: An example of a 3-coloring problem with 6 nodes and 9 edges.

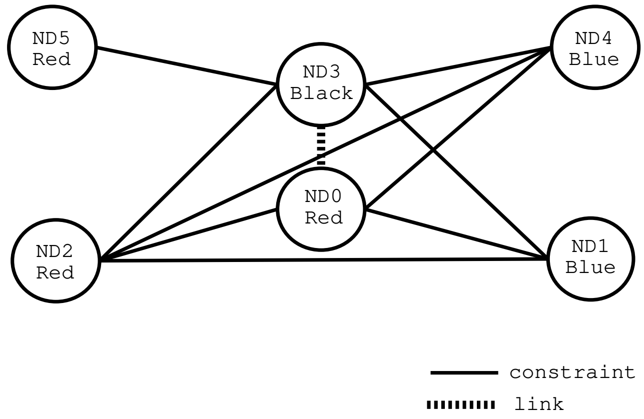

Figure 7: The state of the sample problem after ND3 leads the first mediation.

quest to, 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 agents in the session. Currently, solutions are chosen based on the min-conflict heuristic [12]. Although, not currently implemented, ties can be broken based on the number of variables that would need to change values. Changing fewer variables promotes stability.

Once the solution is chosen, "accept!" messages are sent to the agents in the session, who, in turn, adopt the proposed answer.

3.3 An Example

To further explain how the algorithm works, it helps to have an example. Consider the 3-coloring problem presented in figure 6. In this problem, there are 6 agents, each with a variable and 9 edges or constraints between them. In this problem, the constraint between ND2 and ND3 is in violation because both agents have the color Black assigned to their variables. Following the algorithm, upon startup each agent sends an "init" message to its neighbors. Because each of the incoming "init" messages is from a neighbor, the information is added to the $good\_list$ and each agent computes the set of acceptable solutions for the subgraph that includes itself and its neighbors.

Once the startup has been completed, each of the agents checks its $agent\_view$. ND3 and ND2 find that they are in conflict. ND2, being the lower priority of the two, waits for ND3 to start a negotiation. ND3, knowing it is higher priority, first checks to see if it can resolve the conflict by changing its value, which in this case, it cannot. ND3 starts a negotiation that involves ND1, ND2, ND4, and ND5. It sends each of them and "evaluate?" message. Since ND3 was not able to prune any of the values from the domains of any of the variables in its $good\_list$, each of the agents in the negotiation receives the choice of the three colors.

When each of the agents in the mediation receives the "evaluate?" message, they label each domain element with the names of any variables that they would have a conflict with as a result of adopting that value. The following are the labeled domains for each of the agents

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

As the mediator, ND3, chooses a solution that minimizes the conflict that will be caused, yet resolves its local constraints. In this case, it chooses to have ND2 change its color to Red, introducing a new conflict with between ND2 and ND0. Finally, ND3 links with ND0, leaving the problem in the state shown in figure 7.

ND1, ND2, ND4 and ND5 inform the agents in their $agent\_view$ of their new values, then checks for conflicts. This time, ND2 and ND0 notice that their values are in conflict. ND2 becomes the mediator and negotiates with ND0, ND1, ND3, and ND4. Following the protocol, ND2 sends out the "evaluate?" messages, still with the full domain for each agent, and the receiving nodes label and respond. The following are the labeled domains that are returned:

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

ND2 after receiving these messages, checks its $good\_list$ and finds two solution that solve its subproblem. It chooses to change the colors of all of the nodes it is mediating over to the values in figure 8 and the problem is solved.

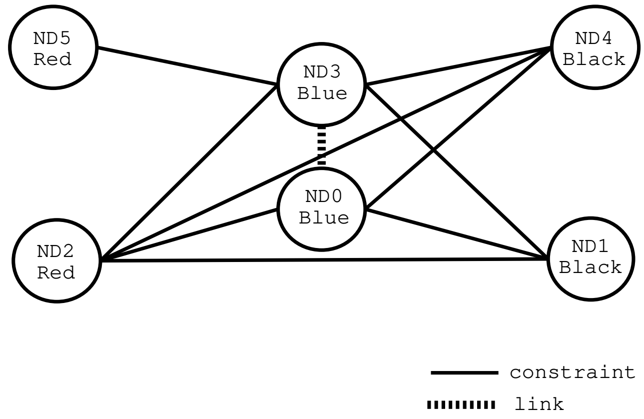

Figure 8: The final solution after ND2 leads the second mediation.

3.4 Soundness and Completeness

In this section we will show that the APO algorithm is both sound and complete. For these proofs we assume that all communications are reliable, meaning that if a message is sent from x_i to x_j that x_j will receive the message in a finite amount of time. We also assume that if a message m_1 is sent before another message m_2, that m_1 will be received before m_2. Before we prove the soundness and completeness, it helps to have a few principle proofs established.

Lemma 1 *Links are bidirectional. i.e. If x_i has x_j in its $agent\_view$ then x_j has x_i in its $agent\_view$.*

Proof:

Assume that x_i has x_j is its $agent\_view$ and that x_i is not in the $agent\_view$ of x_j. In order for x_i to have x_j in its $agent\_view$, x_i must have received and "init" message at some point from x_j. There are two cases.

Case 1: x_j is in the $initList$ of x_i. In this case, x_i must have sent x_j an "init" message first, meaning that x_j received an "init" message and therefore has x_i in its $agent\_view$, a contradiction.

Case 2: x_j is not in the $initList$ of x_i. In this case, when x_i receives the "init" message, it responds with an "init" message. That means that if the reliable communication assumption holds, eventually x_j will receive x_i's "init" message and add x_i to its $agent\_view$. Also a contradiction.

Lemma 2 *If two agents are linked, then their agent_views are kept up to date.*

Proof:

Assume that x_i has a value in its $agent\_view$ for x_j that is not the correct value of x_j. This would mean that at some point x_j set its value without informing x_i. There are two cases:

Case 1: x_j did not know it needed to send x_i an update. i.e. x_i was not in x_j's $agent\_view$. Contradicts lemma 1.

Case 2: x_j did not inform all of the agents in its $agent\_view$ when it changes its value. It is clear from the code that this cannot happen. Agents only change their values in procedures check_agent_view, choose_solution and accept!. In each of these cases, it sends an "ok?" or informs the agents through the "accept" message that a change to its value has occurred.

Lemma 3 *If x_i is in conflict with one or more of its neighbors, does not expect a negotiation from another higher priority agent in its $agent\_view$ and is currently not in a negotiation, then it will act as mediator.*

Proof:

Directly from the procedure check_agent_view.

Lemma 4 *If x_i mediates a negotiation that has a solution, then each of the constraints between the agents involved in the mediation will be satisfied.*

Proof:

Assume that there are two agents x_j and x_k (either of them could be x_i), that were mediated over by x_i and after the mediation there is a conflict between x_j and x_k. There are two ways this could have happened.

Case 1: One or both of the agents must have a value that x_i did not assign to it as part of the mediation.

Assume that x_j and/or x_k has a value that x_i did not assign. To do this, x_j and/or x_k, must have had their negotiation flag set to false, since the only times they can change their value is if that flag is false, they are the mediator, or the mediator tells them to. Also, we know that since x_i mediated a negotiation including x_j and x_k, that x_i did not receive a wait! message from either of them. This leads to a contradiction, because if x_j and x_k sent evaluate! messages, they must have set their negotiate flag to true.

Case 2: x_i assigned them a value that caused them to be in conflict with one another.

Let's assume that x_i assigned them conflicting values. This means that x_i chose a solution from its *good_list* that did not take into account the constraints between x_j and x_k. But, according to definition 3, we know that the *good_list* only contains solutions that include all of the constraints between all of the agents in the *good_list*. This leads to a contradiction.

This lemma is important because it says that once a mediator has concluded its session, that the only conflicts that can exist are on constraints that agents outside of the mediation are part of. This has the effect of pushing the constraint violations outside of the mediators view or to the edge of the constraint graph it has a view of. In addition, because mediators get information about who the violations are being pushed to and establish links with those agents, over time, they gain more context. This is a very important point when considering the completeness of the algorithm.

Theorem 1 *The APO algorithm is sound. i.e. It reaches a stable state iff it has either found an answer of no solution exists.*

Proof:

In order to be sound, the agents can only stop whenever they have reached an answer. The only condition is which they would stop without having found an answer is if one or more of the agents is expecting a negotiation from a higher priority agent that does not negotiate. Let's say we have 3 agents, x_i, x_j, and x_k with $i < j \vee k < j$ (i could be equal to k) and x_k has a conflict with x_j. There are three conditions in which x_j would not mediate a negotiation that include x_i, if x_i was expecting it to.

Case 1: x_j does not recognize that it has a conflicting value with x_k.

Assume that x_j is in conflict with x_k and does not know it. This means that at some point x_k changed its values and x_j does not know about it. Since, as part of initialization, each agent adds each of its neighbors to its $agent\_view$ through an exchange of "init" messages, we know that lemma 2 holds, which leads to a contradiction.

Case 2: x_i believes that x_j has a conflict with x_k when it does not.

Assume that x_i believes that x_j is in conflict with some x_k that is a neighbor to x_j and they are not. In order for this to happen, x_i must have x_k and x_j in its $good\_list$ and an out of date view of their values. Agents are only added to the $good\_list$ after an "init" message has been received, which means that they are also in the $agent\_view$. By lemma 1 and lemma 2 we know that x_i must be getting updated information from x_j and x_k which means that each of them know the other value. Therefore, x_i cannot have an out of date view. A contradiction.

Case 3: x_j does not know it should include x_i in the negotiation.

This case actually doesn't matter. If x_j doesn't know that it should include x_i, then it will conduct its mediation without x_i and send an updated value to x_i at the end. x_i will then know that the conflict no longer exists (the previous case) and will no longer expect a negotiation from x_j.

Definition 4 *Oscillation is a condition that occurs when a subset $V' \subseteq V$ of the agents are infinitely cycling through their allowable values without reaching a solution.*

By this definition, in order to be considered part of an oscillation, an agent within the subset must be changing its value (if it's stable, it's not oscillating) and it must be connected to the other members of the subset by a constraint (otherwise, it is not actually a part of the oscillation).

Theorem 2 *The APO algorithm is complete. i.e. if a solution exists, the algorithm will find it. If a solution does not exist, it it will report that fact.*

Proof:

A solution does not exist whenever the problem is over-constrained. If the problem is over-constrained, our algorithm will eventually produce a $good\_list = \emptyset$ meaning that some subset $V' \subseteq V$ and the associated constraints on those variables lead to no solution. Since a subset of the variables is unsatisfiable, the entire problem is unsatisfiable, therefore, no solution is possible. Our algorithm terminates with failure if and only if an empty $good\_list$ is created.

Since we have now shown in Theorem 1 that whenever our algorithm reaches a stable state, the problem is solved and that when it creates an empty $good\_list$ it terminates, we only need to show that it reaches one of these two states in finite time. The only way for the agents to not reach a stable state is when one or more of the agents in the system is in an oscillation.

There are two cases to consider, the easy case is when a single agent is oscillating ($|V'| = 1$) and the other case is when more than one agent is oscillating ($|V'| > 1$).

Case 1: There is an agent x_i that is caught in an infinite loop and all other agents are stable.

Let's assume that x_i is in an infinite processing loop. That means that no matter what it changes its value to, it is in conflict with one of its neighbors, because if it changed its value to something that doesn't conflict with its neighbors, it would have a solution and stop. If it changes its value to be in conflict with some x_j that is higher priority than it, then x_j will mediate a negotiation with x_i, contradicting the assumption that all other agents are stable. If x_i changes its value to be in conflict with a lower priority agent, then by lemma 3, it will act as mediator with its neighbors. Since it was assumed that each of the other agents is in a stable state, then all of the agents in x_i's $good\_list$ will participate in the negotiation and by lemma 4, agent x_i will have all of its conflicts removed. This means that x_i will be in a stable state contradicting the assumption that it was in an infinite loop.

Case 2: Two or more agents are in an oscillation.

Let's say we have a set of agents $V' \subseteq V$ that are in an oscillation. Now consider an agent x_i that is within V'. We know that the only conditions in which x_i changes its value is when it can do so and solve all of its conflicts (a contradiction because it wouldn't be considered part of the oscillation), as the mediator of a negotiation, or as the receiver of a mediation from some other agent in V'. The interesting case is when an agent acts as the mediator.

Consider the case when x_i is the mediator and call the set of agents that it is mediating over V_i. We know according to definition 4 that after the mediation, that at least one conflict must be created or remain otherwise the oscillation would stop and the problem would be solved. In fact, we know that each of the remaining conflicts must contain an agent from the set $V' - V_i$ by lemma 4. We also know that for each violated constraint that has a member from V_i, that x_i will link with any agent that is part of those constraints and not a member of V_i. The next time x_i mediates, the set V_i will include these members and the number of agents in the set $V' - V_i$ is reduced. In fact, whenever x_i mediates the set $V' - V_i$ will be reduced (assuming it is not told to wait! by one or more agents. In this case, it takes longer to reduce this set, but the proof still holds). Eventually, after $O(|V|^2)$ mediations, some x_i within V' must have $V_i = V'$ (every agent within the set must have mediated $|V'|$ times in order for this to happen). When this agent mediates it will push the violations outside of the set V' or it will solve the subproblem by lemma 4. Either of these conditions contradicts the oscillation assumption. Therefore, the algorithm is complete.

It should be fairly clear that in order to be complete, the algorithm runs in exponential time. In addition, the space complexity of the algorithm is also exponential, because in the worst case one or more of the agents will centralize the entire problem.

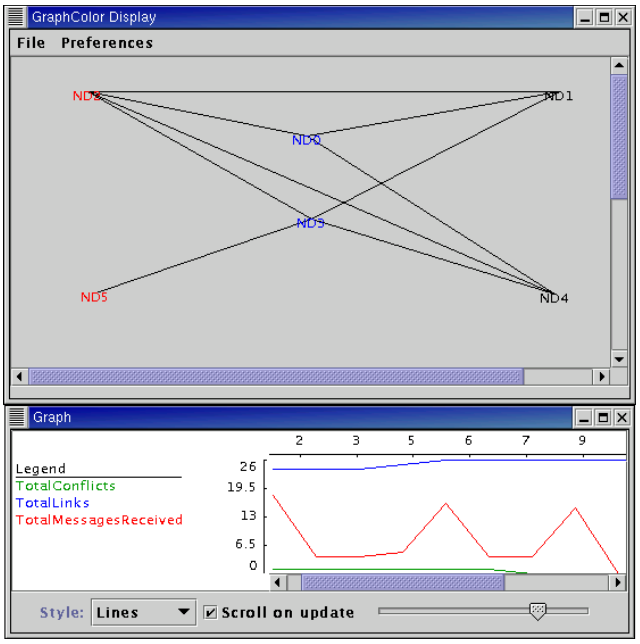

Figure 9: The displays used for the graph coloring domain in the Farm for the example problem.

4 Evaluation

To test the APO algorithm, we implemented the AWC and APO algorithms and conducted experiments in the distributed 3-coloring domain. The particular AWC algorithm we implemented can be found in [20] which does not include the resolvent *nogood* learning mechanism described in [5]. The distributed 3-coloring problem is a 3-coloring problem with n variables and m binary constraints distributed amongst the agents. We created solvable graph instances with $m = 2.0n$ (considered *sparse*) and $m = 2.7n$ (considered *critical*) [2] according to the method presented in [12]. We generated 10 random graph for $n = 15, 30, 45$ and for each instance generated 10 initial variable assignments. In total for each combination of n and m, we ran 100 trials.

The algorithms were tested in a partially asynchronous simulation environment called the Farm (see figure 9)[6]. In this environment, each of the agents is run as a separate thread and allowed to complete one *cycle* of execution. During each cycle, incoming messages are delivered, the agent is allowed to process the information, and any messages that were created during the processing are added to the outgoing queue to be delivered at the beginning of the next cycle. This means that information sent by one agent during this cycle is not available to the receivers till the next cycle. Agents are given processing time in random order and when distributed on different computers are executed in parallel. The actual execution time given to one agent during a cycle varies according to the amount of work needed to process all of the incoming messages.

To evaluate the relative strengths and weakness of each of the approaches, we measured the number of *cycles*, the number of messages, and the number of *links* that were established

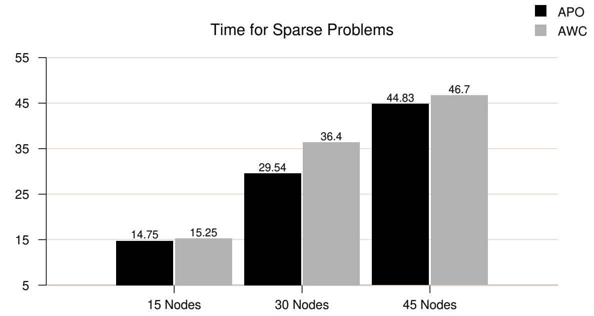

Figure 10: Comparison of the time (measured in cycles) needed to solve random sparse 3-coloring problems of various size by AWC and APO.

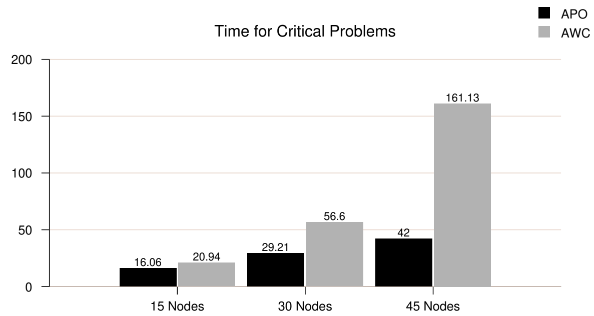

Figure 11: Comparison of the time (measured in cycles) needed to solve random critical 3-coloring problems of various size by AWC and APO.

during the course of solving each of the problems. The random seeds used to create each graph instance and variable instantiation were saved and used by each of the algorithms for fairness.

As you can see from the results presented in figures 10 and 11, the APO algorithm does better on average than the AWC algorithm on both sparse and critical problems. In fact, APO does only slightly worse on critical problems than it does on sparse ones, but considerably better than AWC. This is most likely caused by the fact that in critical problems, the mediating agents both more quickly learn about other agents along the critical paths of the problem and that they are able to remove large parts of the search space from their $good\_lists$ based on the criticalness of the problem. AWC, on the other hand, has to generate, through trial and error, a substantial number of *nogoods* and links in order to come to the same conclusion.

The most profound differences between the two techniques is the number of messages that are needed to solve the problem and the link density that results from their execution. APO does better in all cases in both categories. This is probably caused by the need for AWC to discover, through trial and error, the interdependencies that exist with non-local agents. APO is given these interdependencies and is able to remove

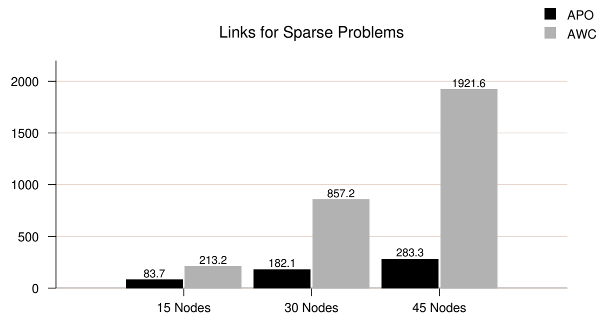

Figure 12: Comparison of the number of links established while solving random sparse 3-coloring problems of various size by AWC and APO.

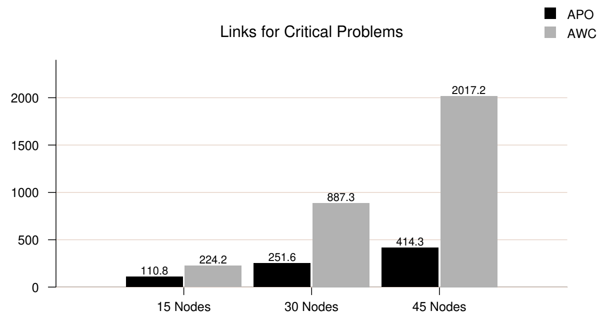

Figure 13: Comparison of the number of links established while solving random critical 3-coloring problems of various size by AWC and APO.

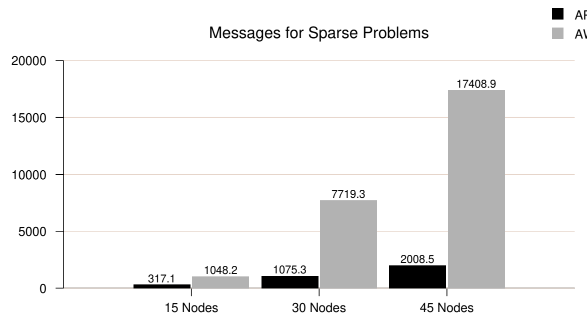

Figure 14: Comparison of the number of messages sent while solving random sparse 3-coloring problems of various size by AWC and APO.

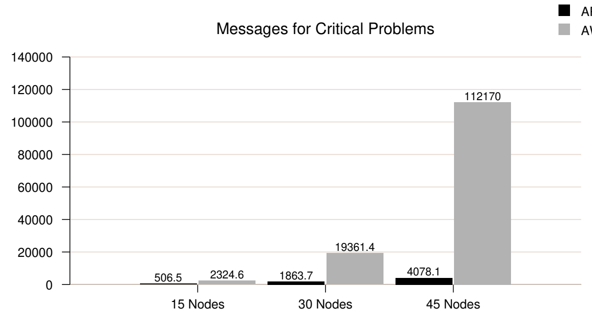

Figure 15: Comparison of the number of messages sent while solving random critical 3-coloring problems of various size by AWC and APO.

a vast number of *nogood* solutions simple through the local partial searches being conducted in the creation of the $good\_list$.

5 Conclusions and Future Work

In this paper, we presented a new method for solving DisCSPs called the *Asynchronous Partial Overlay* algorithm. 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 of the constraint graph to improve their decisions. We have shown that the APO algorithm is both sound and complete and that it performs as well as, if not better than the AWC algorithm for both sparse and critical graph coloring problems.

Future work on this protocol includes evaluation of the protocol in both additional CSP domains (such as SAT) and against additional DisCSP techniques. In addition, there are currently several proposed extensions to the protocol that would allow it to work in dynamic environments and on optimization problems [10; 11; 15]. We are also considering making a parameterized version of the protocol called APO(λ) that would allow us to vary the centralization that occurs in the protocol. In this algorithm, λ would control the distance that an agent checks for conflict when trying to decide when to mediate. So, for example, the current APO algorithm is APO(1), meaning that agents mediate over problems with their neighbors. In APO(2), agents could mediate over conflicts at distance 2 from themselves, etc. It is our belief that changing the amount of centralization effects the convergence of the protocol.

Acknowledgments

The authors would like to thanks Bryan Horling for his efforts on the Farm simulation environment and for his implementation of the AWC algorithm.

References

[1] Bernard Burg. Parallel forward checking: Second part. Technical Report TR-595, Institute for New Generation Computer Technology, September 1990.

[2] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, 1991.

[3] S. E. Conry, K. Kuwabara, V. R. Lesser, and R. A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6), November 1991.

[4] Rina Dechter. Constraint processing incorporating backtracking, learning, and cluster-decomposition. In *Proceedings of the Fourth IEEE Conference on Artificial Intelligence for Applications (CAIA)*, pages 312–319, March 1988.

[5] Katsutoshi Hirayama and Makoto Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *The 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 169–177, 2000.

[6] Bryan Horling, Roger Mailler, and Victor 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)*, May 2003.

[7] Victor R. Lesser and Daniel D. Corkill. The distributed vehicle monitoring testbed. *AI Magazine*, 4(3):63–109, Fall 1983.

[8] Qiangyi Luo, P. G. Hendry, and J. T. Buchanan. A hybrid algorithm for distributed constraint satisfaction problems. In *Proceedings of the Ninth European Workshop on Parallel Computing (EWPC)*, Barcelona, Spain, 1992.

[9] Qiangyi Luo, P. G. Hendry, and J. T. Buchanan. Heuristic search for distributed constraint satisfaction problem. Technical Report KEG-6-93, University of Strathclyde, Strathclyde, Scotland, 1993.

[10] Roger Mailler, Victor Lesser, and Bryan Horling. Cooperative negotiation for soft real-time distributed resource allocation. In *In Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2003)*, Melbourne, AUS, 2003. Also submitted as UMass Computer Science Technical Report 2002-49.

[11] Roger Mailler, Regis Vincent, Victor Lesser, Tim Middlekoop, and Jiaying Shen. Soft-real time, cooperative negotiation for distributed resource allocation. *AAAI Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems*, November 2001.

[12] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[13] Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, and Shriniwas Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In John-Jules Meyer and Milind Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 181–193, 2001.

[14] K. Sycara, S. Roth, N. Sadeh, and M. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1446–1461, November/December 1991.

[15] Guandong Wang, Weixiong Zhang, Roger Mailler, and Victor Lesser. *Analysis of Negotiation Protocols by Distributed Search*. 2003. To appear.

[16] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP-95)*, Lecture Notes in Computer Science 976, pages 88–102. Springer-Verlag, 1995.

[17] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[18] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

[19] Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *International Conference on Multi-Agent Systems (ICMAS)*, 1996.

[20] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):198–212, 2000.