Using Cooperative Mediation to Solve Distributed Constraint Satisfaction Problems*

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

Abstract

Distributed Constraint Satisfaction (DCSP) has 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 constraint satisfaction and partly because real-world problems often present themselves in a distributed form. In this paper, we present a complete, distributed algorithm called asynchronous partial overlay (APO) for solving DCSPs that is based on a cooperative mediation process. The primary ideas behind this algorithm are that agents, when acting as a mediator, centralize small, relevant portions of the DCSP, that these centralize subproblems overlap, and that agents increase the size of their subproblems along critical paths within the DCSP as the problem unfolds. We present empirical evidence that shows that APO performs better than other known, complete DCSP techniques.

1. Introduction

Distributed constraint satisfaction has become a classic formulation that is used to describe a number of distributed problems including distributed resource allocation [1], distributed scheduling [8], and distributed interpretation [5]. It's no wonder that a vast amount of effort and research has gone into creating algorithms, such as distributed breakout (DBO) [10], asynchronous breakout (ABT) [9], and asynchronous weak-commitment (AWC) [11], for solving these problems.

Unfortunately, a common drawback to each of these techniques is that they prevent the agents from making informed local decisions about the effects of changing their local variable value without actually doing it. For example, in AWC, agents try to a value and wait for another agent to tell them that it will not work through a nogood message. Because of this, agents never learn why another agent or set of agents is unable to accept the value, they only learn that their value in combination with other values doesn't work.

In this paper, we present a cooperative mediation based DCSP protocol, called Asynchronous Partial Overlay (APO), that allows the agents to extend and overlap the context that they use for making their local decisions. 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 technique allows for rapid, distributed, asynchronous problem solving without the explosive communications overhead normally associated with current distributed algorithms. APO represents a new methodology that lies somewhere between centralized and distributed problem solving which exploit the best characteristics of both. In the graph coloring domain, this algorithm performs better, both in terms of communication and computation, than the AWC algorithm. This is particularly true for problems that lie near or to the right of the phase transition.

In the rest of this paper, we present a formalization of the DCSP problem. We then present the APO algorithm and present an example of the execution on a simple problem. Next, we present the results of extensive

* 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 notation thereon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
testing that compares APO with AWC within the commonly used graph coloring domain. Lastly, we discuss some of our conclusions and future directions.

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 $R_i(d_1, \ldots, d_j)$ is a predicate on the Cartesian product $D_1 \times \cdots \times D_j$ that returns true if the value assignments of the variables satisfy 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. CSP has been shown to be NP-complete, making some form of search a necessity.

In the distributed case (DCSP), 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 is satisfied. Clearly, each agent’s 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 the value of its variable.

when received $(ok?, (x_j, p_j, d_j, m_j))$ do
update agent.view with $(x_j, p_j, d_j, m_j);$
check_agent.view;
end do;

procedure check_agent.view
if initList ≠ 0 or mediate ≠ false do
return;
m' ← hasConflict(x_i);
if m' and ¬3p_i > p_j ∧ m_j = true
and conflicts are with lower priority neighbors
d_i ← d'_i;
send $(ok?, (x_j, p_j, d_j, m_j))$ to all $x_j \in agent.view;$
else
do mediate;
else if m_i ≠ m'_i
m_i ← m'_i;
send $(ok?, (x_j, p_j, d_j, m_j))$ to all $x_j \in agent.view;$
end if;
end check_agent.view;

Figure 2. The procedures for doing local resolution, updating the agent.view and the goodList.

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 which are of the form $R_i(x_1, x_2)$. It is fairly easy to extend our approach to handle the case where one or both of the restrictions are removed.

3. Asynchronous Partial Overlay

3.1. The Algorithm

Figures 1, 2, 3, 4, and 5 present the basic APO algorithm. The algorithm works by constructing a goodList and maintaining a structure called the agent.view. The agent.view holds the names, values, domains, and constraints of variables to which an agent is linked. The goodList holds the names of the variables that are known to be connected to the owner by a path in the constraint graph.

As the problem solving unfolds, each agent tries to solve the subproblem it has centralized within its goodList or determine that it is unsolvable which indicates the entire global problem is over-constrained. To do this, agents take the role of the mediator and attempt to change the values of the variables within the mediation session to achieve a satisfied subsystem. When this cannot be achieved without causing a violation for agents outside of the session, the mediator links with those agents assuming that they are somehow related to the mediator’s variable. This process continues un-
procedure mediate
    preferences ← 0;
    counter ← 0;
    for each $x_i \in \text{goodList}$ do
        send(evaluate? $\{x_i, p_i\}$ to $x_j$;
        counter + 1;
    end do;
    mediate ← true;
end mediate;

when receive [wait!] $\{x_i, p_i\}$ do
    update agentview with $\{x_i, p_i\}$;
    counter ← 1;
    if counter == 0 do choose solution;
end do;

when receive [evaluate] $\{x_j, p_j, labeled D_j\}$ do
    record $\{x_j, labeled D_j\}$ in preferences;
    update agentview with $\{x_j, p_j\}$;
    counter ← 1;
    if counter == 0 do choose solution;
end do;

Figure 3. The procedures for mediating a session.

procedure choose solution
    select a solution $s$ using a Branch and Bound search that:
    1. satisfies the constraint $c$ of agents in the $\text{goodList}$
    2. minimizes the violations for agents outside of the session
    if $\exists s$ that satisfies the constraints do
        broadcast solution
        for each $x_j \in \text{agent view}$ do
            if $x_j \in \text{preferences}$
                if $d_k \in s$ violates $a_k \in x_j$ and $x_k \notin \text{agent view}$ do
                    send(init $\{x_j, p_i, d_i, m_i, C_i\}$ to $x_j$);
                    add $x_k$ to initlist;
                end if;
                else
                    send(ok $\{x_j, p_i, d_i, m_i\}$ to $x_j$);
                end if;
            end if;
        end do;
        mediate ← false;
        check agentview;
end choose solution;

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

til one of the agents finds an unsatisfiable subsystem, or all of the conflicts have been removed.

In order to facilitate the problem solving process, each agent has a dynamic priority that is based on the size of their $\text{goodList}$ (if two agents have the same sized $\text{goodList}$ then the tie is broken using the lexicographical ordering of their names). Priorities are used by the agents to decide who mediates a session when a conflicts arises. Priority ordering is important for two reasons. First, priorities ensure that the agent with the most knowledge gets 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.

3.1.1. Initialization (Figure 1) On startup, the agents are provided with the value (they pick it randomly if one is not assigned) and the constraints on their variable. Initialization proceeds by having each of the agents send out an “init” message to its neighbors. This initialization message includes the variable’s name ($x_i$), priority ($p_i$), current value ($d_i$), the agent’s desire to mediate ($m_i$), domain ($D_i$), and constraints ($C_i$). The array $\text{initList}$ 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 $\text{agentview}$ and adds the variable to the $\text{goodList}$ if it can. An variable is only added to the $\text{goodList}$ if it is a neighbor of another variable already in the list. This ensures that the graph created by the variables in the $\text{goodList}$ always remains connected, which focuses the agent’s internal problem solving on variables which it knows it has an interdependence with. The $\text{initList}$ is then checked to see if this message is a link request or a response to a link request. If an agent is in the $\text{initList}$, it means that this message is a response, so the agent removes the name from the $\text{initList}$ and does nothing further. If the agent is not in the $\text{initList}$ then it means this is a request, so a response “init” is generated and sent.

It is important to note that the agents contained in the $\text{goodList}$ are a subset of the agents contained in the $\text{agentview}$. This is done to maintain the integrity of the $\text{goodList}$ 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 receive end of the link request has to store the name of the requestor, but cannot add them to their $\text{goodList}$ until a path can be identified.

3.1.2. Checking the agent view (Figure 2) After the agents receive all of the initialization messages they are expecting, they execute the check agentview procedure. In this procedure, the current $\text{agentview}$ (which contains the assigned, known variable values) is checked to identify conflicts between the variable owned by the agent and its $\text{neighbors}$. If, during this check, an agent finds a conflict with one or more of its neighbors and has not been told by a higher priority agent that they want...
to mediate, it assumes the role of the mediator.

An agent can 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 recompotes the value of this flag based on whether or not it has existing conflicts with its neighbors. When this flag is set to `true` it indicates that the agent wishes to mediate if it is given an opportunity. This mechanism acts like a tw-o-phase commit protocol, commonly seen in database systems, and ensures that the protocol is live-lock and dead-lock free.

As the mediator, an agent first attempts to rectify the conflict(s) by changing its own variable. This simple, but effective technique prevents sessions from occurring unnecessarily, which stabilizes the system and saves message and time. If the mediator finds a value that removes the conflict, it makes the change and sends out an "ok" message to the agents in its `agent\_view`. If it cannot find a non-conflicting value, it starts a mediation session. An "ok" message is similar to an "init" message, in that it contains information about the priority current value, etc. of a variable.

### 3.1.3. Mediation (Figures 3, 4, and 5)

The most complex and certainly most interesting part of the protocol is the mediation. 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 session request from a higher priority agent. The mediation starts with the mediator sending out "evaluate?" messages to each of the agents in its `good\_list`. The purpose of this message is tw-o-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 `mediate` in the figures, prev ents thangent from engaging in tw-o sessions 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. This allows the mediator to understand the greater impact of its decision and is also used to determine how to extend its view once it makes its final decision.

When an agent receives a mediation request, it will respond with either a “wait!” or “evaluate!” message. The “wait” message indicates to the requester that the agent is currently involved in a session or is expecting a request from a agent of higher priority than the requester. If the agent is available, it labels each of its domain elements with the names of the agents that it would be in conflict with if it were asked to take e that value. In the graph coloring domain, the labeled domain can never exceed \( O(|D_i| + n) \). This information is returned in the “ev aluate!” message. It should be noted that the agents need not return all of the names if for security reasons they wish not to. This may effect the completeness of the algorithm, because, in the worst case, the completeness relies on one or more of the agents eventually centralizing the entire problem, but does provides some degree of autonomy and privacy to the agents.

When the mediator has received either a “wait!” or “ev aluate!” message from all of the agents that it has sent a request to (which it determines by using the `counter` variable), 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 generated using a Branch and Bound search [3] where all of the constraints must be satisfied and the number of outside conflicts is minimized (like the min-conflict heuristic [7]). The search is also done using the current value assignments as the first branch in the search. These heuristics, when combined together, form a lock and key mechanism that simultaneously exploits the work that was previously done by other mediators and acts to minimize the number of changes in those assignments. As will be presented in section 4, these simple feed-forward mechanisms, combined with the limited centralization needed to solve satisfiability problems, account for considerable improvements in the algorithms runtime performance.

When no satisfying assignments are found, the agent announces that the problem is unsatisfactory and the algorithm terminates. Once the solution is chosen, “accept!” messages are sent to the agents in the session, who, in turn, adopt the proposed answer.

The mediator also sends “ok” messages to the agents
3.2. An Example

Consider the 3-coloring problem presented in figure 6(a). In this problem, there are 8 agents, each with a variable and 12 edges or constraints between them. Because this is a 3-coloring problem, each variable can only be assigned one of the three available colors (Black, Red, or Blue). The goal is to find an assignment of colors to the variables such that no two variables, connected by an edge, have the same color.

In this example, three constraints are in violation: (ND0, ND1), (ND1, ND3), and (ND6, ND7). Following the algorithm, upon startup each agent adds itself to its good list and sends an “init” message to its neighbors. Upon receiving these messages, the agents add each of their neighbors to their good list because they are able to identify a shared constraint with themselves.

Once the startup has been completed, each of the agents checks its agent view. ND0, ND1, ND3, ND6, and ND7 find that they have conflicts. ND0 (priority 3) waits for ND1 to mediate (priority 5). ND6 and ND7, both in priority 4, wait for ND3 to start a mediation. ND3, having an equal number of agents in its good list, but a lower lexicographical order, also waits for ND3 to start a mediation.

ND3, knowing it is higher priority, first checks to see if it can resolve its conflict by changing its value, which in this case, it cannot. ND3 starts a session that involves ND1, ND5, ND6, and ND7. It sends each of them an “ev aluate?” message.

When each of the agents in the mediation receives the “ev aluate?” message, they label their domain elements with the names of the variables that they would be in conflict with as a result of adopting that value. They each send ND3 an “ev aluate?” message with this information. The following are the labeled domains for each of the agents:

- ND1 - Black conflicts with ND2; Red conflicts with ND0 and ND3; Blue conflicts with ND4
- ND5 - Black causes no conflicts; Red conflicts with ND3; Blue causes no conflicts
- ND6 - Black conflicts with ND7; Red conflicts with ND3; Blue conflicts with ND4
- ND7 - Black conflicts with ND6; Red conflicts with ND3; Blue conflicts with ND4

Once all of the responses are received, the mediator, ND3, conducts a branch and bound search that attempts to find a satisfying assignment to the problem that minimizes the amount of conflict that would be created outside of the mediation. If it cannot find at least one satisfying assignment, it broadcasts that a solution cannot be found. In the example, it chooses change ND5’s color to Red, ND7’s color to Red, and its own color to Blue. ND3 cannot solve the conflict between ND0 and ND1.
by making these changes, so it links with ND0, leaving the problem in the state shown in figure 6(b). Note that when this happens, ND3 adds ND0 to its _good_list_ and vice versa.

ND1, ND5, ND6 and ND7 inform the agents in their _agent_view_ of their new values, then check for conflicts. This time, ND0 and ND1 notice that their values are in conflict. ND1, having a higher priority, becomes the mediator and mediates a session with ND0, ND2, ND3, and ND4. Following the protocol, ND1 sends out the “evaluate” messages and the receiving agents label and respond. The following are the labeled domains that are returned:

- **ND0** - Black conflicts with ND2; Red conflicts with ND1; Blue causes no conflicts
- **ND2** - Black cause no conflicts; Red conflicts with ND0 and ND1; Blue conflicts with ND4
- **ND3** - Black conflicts with ND6; Red conflicts with ND1, ND5, and ND7; Blue causes no conflicts
- **ND4** - Black conflicts with ND2 and ND6; Red conflicts with ND1 and ND7; Blue causes no conflicts

ND1, after receiving these messages, conducts its search and solves the problem and returns the solution subproblem. It chooses to change the color of ND0 to Blue. ND0, ND1, ND2, ND3, and ND4 check their _agent_view_ and find no conflicts, so the problem is solved (see figure 6(c)).

### 3.3. Soundness and Completeness

The proofs of APO’s soundness and completeness are quite lengthy, so for simplicity, we refer the reader to [6] for their full details. Below are the main ideas that are used in them.

- If at anytime an agent identifies a constraint subgraph that is not satisfiable, it announces that the problem cannot be solved. Half of the soundness.
- If a constraint violation exists, someone will try to fix it. The protocol is dead-lock free. The other half of the soundness.
- Even trivially in the worst case, one or more of the agents will centralize the entire problem and will derive a solution, or report that no solution exists. This ensures completeness.

### 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 [11] which includes the resolvemtc no_ good learning mechanism described in [4]. We conducted 3 sets of experiments.

In the first set of experiments, following the experimental setup presented in [11], we created solvable graph instances with \( m = 2.0n \) (low-density) and \( m = 2.7n \) (high-density) according to the method presented in [7]. We generated 10 random graph for \( n = 15, 30, 45, 60, 75, 90 \) and for each instance generated 10 initial variable assignments. For each combination of \( n \) and \( m \), we ran 100 trials making a total of 1800 trials. During this series, we measured the number of messages and **cycles** used by the algorithms. During a cycle, incoming messages are delivered, the agent allows 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. 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. The results from this experiment can be seen in figures 7 and 8.

In the second set of experiments, we created completely random 60 node graphs of various densities from 1.8 to 2.9. This was done to test the completeness of the algorithms and to verify the correctness of their implementations. For each density value, we generated 200 random graphs each with a single set of initial values. Again, we measured the number of cycles and the number of messages used by each of the algorithms. In total,
1400 graphs were generated and tested. We stopped the execution of the algorithms at 1000 cycles for the sake of time. The results of these experiments are shown in figures 9(a) and 9(b).

In the third set of experiments, we directly compared the serial run time performance of A WC against APO. For these experiments, we again generated random graphs, this time varying the size and the density of the graph. We generated 25 graphs for the values of $n = 15, 30, 45, 60$ and the densities of $d = 2.0, 2.3, 2.7$, for a total of 300 test cases. To show that the performance difference in APO and AWC was not caused by the speed of the central solver, we ran a centralized backtracking algorithm on the same graph instances. Although, APO uses the branch and bound algorithm, the backtracking algorithm used in this test provides a best case lower bound on the runtime of APO’s internal solver.

Each of the programs used in this test was run on an identical 2.4GHz Pentium 4 with 768 MB of RAM. These machines where entirely dedicated to the tests so there was a minimal amount of interference by competing processes. In addition, no computational cost was assigned to message passing because the simulator passes messages between cycles. The algorithms were, how ever, penalized for the amount of time they took to process messages. Although we realize that the specific implementation of an algorithm can greatly affect its runtime performance, every possible effort was made to optimize the AWC implementation used for these experiments in an effort to be fair.

For satisfiable graph instances, you can see that on low-density satisfiable graphs A WC and APO perform almost identically in terms of cycles to completion. When the density of the graphs increase to 2.7 how ever, the difference becomes remarkable. APO begins to scale more efficiently than A WC. This can be attributed to the ability of APO to rapidly identify strong interdependencies between variables and to derive solutions to them using a centralized search of the partial subproblem. We should mention that the results of the testing on A WC obtained from these experiments agree with those reported in [4] verifying the correctness of our implementation.

On random, 60 variable instances, APO significantly outperforms A WC on all but the simplest of problems (see figure 9(a)). The most direct cause of this is AWC’s poor performance on unsatisfiable problem instances as previously reported in [2].

In the serial run time tests, presented in figures 10(a), 10(b), and 10(c), we also see that APO outperforms A WC. You should note that the scale used for these graphs is logarithmic. In fact, APO actually outperforms the centralized solver on graphs larger than 30 nodes.
5. Conclusions

In this paper, we presented a new method for solving DCSPs 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 better than the AWC algorithm on graph coloring problems of various sizes and difficulty.

References


