Comparing Two Approaches to Dynamic, Distributed Constraint Satisfaction*

Roger Mailler
Cornell University
5160 Upson Hall
Ithaca, NY 14853
rmailler@cs.cornell.edu

ABSTRACT

It is now fairly well understood that a vast number of AI problems can be formulated as Constraint Satisfaction Problems (CSPs) and striking improvements have been made in solving them using both centralized and distributed methods. However, many real world problems change over time and very little work has been done in developing methods, particularly distributed ones, for solving problems which exhibit this behavior.

This paper presents two new protocols for solving dynamic, distributed constraint satisfaction problems which are based on the classic Distributed Breakout Algorithm (DBA) and the Asynchronous Partial Overlay (APO) algorithm. These two new algorithms are compared on a broad class of problems varying the problems' overall difficulty as well as the rate at which they change over time. The results indicate that neither of the algorithms complete dominates the other on all problem types, but that depending on environmental conditions and the needs of the user, one method may be preferable over the other.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Dynamic, Distributed Constraint Satisfaction, Cooperative Mediation, Dynamic Partial Centralization

*Research supported by the Intelligent Information Systems Institute, Cornell University (AFOSR Grant FA9550-04-1-0151)

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.
AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.
Copyright 2005 ACM 1-59593-094-9/05/0007 ...$5.00.

1. INTRODUCTION

Over the past 15 years, tremendous strides have been made to improve the efficiency and reach of both centralized and distributed constraint satisfaction problem (CSP) solvers. Centralized problem solvers can now handle problems with millions of variables and clauses and distributed solvers can solve problems with hundreds of variables and clauses.

Unfortunately, with very few exceptions, little work has been done on solving CSPs which change over time. There are a number of reasons why a CSP may change over time. For example, the CSP may represent a scheduling problem in which users can insert and remove new events or it may be a airspace deconfliction problem where new objects enter and leave the airspace as time goes by.

The earliest mention of this class of problems can be found in [2] where it was first referred to as dynamic CSP (DynCSP). Loosely defined, a DynCSP is a sequence of CSPs which differ from one another by the addition (restriction) or removal (relaxation) of a constraint.

Since then, several techniques have been developed for handling dynamics using centralized solvers. For example in [13], the authors present a method for solving DynCSPs which utilizes local changes to the preexisting variable values. Another method, which can be found in [11] maintains state information about the removal of a possible solution. In this work, when a nogood is created, it is annotated with a justification. Justifications are composed of the subset of constraints which are violated by the values contained within nogood. Because of this, when a constraint is removed, it is easy to identify the nogoods which no longer apply and the solution can be repaired. Lastly, hill-climbing methods such as GSAT [12], can be used to solve DynCSPs because they are designed to repair previously derived solutions.

Following from the work in centralized DynCSPs, two methods have evolved within the realm of dynamic, distributed constraint satisfaction (DynDCSP). The first method is a distributed hill-climbing technique called Peer-to-Peer (P2P) [3]. P2P works by having each agent find the best value for its variable based on its constraints and the values of the variables it shares constraints with. If this value is different from the variable's current value, it changes to the new value with some low probability, then sends out update messages. Overall the behavior is very close to that of centralized hill-climbing techniques.

The second method, which was presented in [9], is an adaptation of the Asynchronous Weak Commitment (AWC)

protocol [14]. In this work, agents annotate their nogoods with the current value of their variable when they are stored. This has a similar effect to the annotation done in centralized problem solving. The key difference, however, is that because the constraints are not explicitly known to the agents, the technique is limited to only handling local constraint changes such as the addition or removal of a domain element. This is why it the algorithm is called Locally Dynamic AWC (LD-AWC).

This paper presents and compares two new methods for solving DynDCSPs. The first is an adaptation of the Distributed Breakout Algorithm (DBA) [15] and is therefore called DynDBA. Like its static counterpart, DynDBA works by making local adaptations to the variable values and is therefore an incomplete search technique. This means that DynDBA continuously works to find minimal conflicting solutions even when the problem is unsatisfiable.

The second algorithm is a modified version of the Asynchronous Partial Overlay (APO) algorithm [6] and is called DynAPO. DynAPO is quite different from DynDBA in a number of ways. The most profound is that, like APO, it is a complete search technique and as such is able to report when it has determined that a problem is unsatisfiable. Because of this, DynAPO often enters period of quiescence when the problem is satisfied or has been determined to be unsatisfiable. The side effect of this, though, is that DynAPO lacks the pure minimization flavor of DynDBA.

To compare these two algorithms, an adaptation of distributed graph coloring was developed which allows for testing problems of various difficulty and that exhibit different rates of dynamics. The results of this testing show that neither of these techniques completely dominates the other in all cases. These results seem to indicate the need for techniques which adapt their behavior not only based on the difficulty of the problem, but on how fast the problem is changing as well.

In the rest of this paper, we will present a formalization of the dynamic distributed constraint satisfaction problem. Unlike other formalizations of this problem, this formalization incorporates a function which dictates the rate at which the constraints change. This explicit connection with time means that changes in the problem's structure are not viewed as episodic (changing only between executions of the problem solvers), but occur continuously while the algorithms attempt to solve them. In sections 3 and 4, the DynDBA and DynAPO algorithms will be presented. In section 5, the experimental setup and results of the testing will be discussed. Section 6 gives conclusion on this work and presents some future directions.

2. DYNAMIC, DISTRIBUTED CONSTRAINT SATISFACTION

A *static* Constraint Satisfaction Problem (CSP), $P = \langle V, D, R \rangle$, 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 $R_i(d_{i1}, \ldots, d_{ij})$ is a predicate on the Cartesian product $D_{i1} \times \cdots \times D_{ij}$ that returns true iff the values assigned to 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. CSP has been shown to be NP-complete, making some form of search a necessity.

Further extending the problem to the dynamic case, a Dynamic CSP is a structure $\langle P_{initial}, \Delta \rangle$ where $P_{initial}$ is a initial CSP and $\Delta : P_t \mapsto P_{t+1}$ is a function which maps the CSP at time t to a new CSP at time $t+1$ by adding and removing constraints. We can measure the amount of change from one instance of a CSP to another by counting the number of added and removed constraints. We can further define a rate function $\delta = \Delta'$, the first derivative of Δ, which defines how fast the dynamic CSP transitions from one static CSP to the other with relation to time. In this paper, we consider Δ to be a linear function which makes δ a constant.

This definition differs from the classic formulation presented in [2] in only one important. Namely, it introduces the function Δ. The Δ function is important because it allows us to measure the usefulness of a particular approach to solving a problem in relation to how fast the problem changes over time. As the rate, δ, is increased, more reactive (and potentially less optimal) problem solving is expected to be more useful. In fact, it is sometimes the case that the problem changes faster than the problem solvers can keep up.

In the distributed case (DynDCSP), 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 the constraints on its variable 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.

In this paper, for the sake of clarity, we restrict ourselves to cases where each agent is assigned a single variable and is given knowledge of the constraints and changes to the constraints on their 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 $R_i(x_{i1}, x_{i2})$. Both of these protocols can be extended to handle cases where one or both of these restrictions are removed. In addition, throughout the paper, we use the term *neighbor* to refer to variables which share a constraint and the *constraint graph* to mean the graph formed by representing the variables as nodes and the constraints as edges.

3. DYNAMIC, DISTRIBUTED BREAKOUT ALGORITHM

3.1 Distributed Breakout

The Distributed Breakout Algorithm (DBA) [15] is an distributed adaptation of the centralized Breakout algorithm [10]. DBA works by alternating between two modes. The first mode (see figure 1) is called the *wait_ok?* mode. During this mode, the agent collects *ok?* messages from each of its neighbors. Once this has happened, the agent calculates the best new value for its variable along with the improvement in its local evaluation. The agent then send out *improve?* message to each of its neighbors then changes to the *wait_improve?* mode. In the *wait_improve* mode (see

```
when received (ok?, x_j, d_j) do
  if mode == wait_improve
    add message to queue;
    return;
  else
    add (x_j, d_j) to agent_view;
    when received ok? messages from all neighbors do
      send_improve;
      mode ← wait_improve;
    end do;
  end if;
end do;

procedure send_improve
  current_eval ← evaluation value of current_value;
  my_improve ← possible maximum improvement;
  new_value ← the value which yields the
        best improvement;
  send (improve, x_i, my_improve, current_eval)
        to neighbors;
end send_improve;
```

Figure 1: The procedures of the *wait_ok?* mode in Distributed Breakout.

```
when received (improve, x_j, improve, eval) do
  if mode == wait_ok
    add message to queue;
    return;
  else
    record message;
    when received improve messages from all
         neighbors do
      send_ok;
      clear agent_view;
      mode ← wait_ok;
    end do;
  end if;
end do;

procedure send_ok
  if my_improve is better than all of my neighbors
    current_value ← new_value;
  end if;
  when in a quasi-local-minimum do
    increase the weights on all violated constraints;
  end do;
  send (ok?, x_i, current_value) to neighbors;
end send_ok;
```

Figure 2: The procedures of the *wait_improve?* mode in Distributed Breakout.

figure 2), the agent collects *improve?* messages from each of its neighbors. Once all of the messages have been received, the agent checks to see if its improvement is the best amongst its neighbors. If it is, it changes its value to the new improved value. If the agent believes it is in a quasi-local-minimum (QLM), it increase the weights on all the of violated constraints. Finally, the agent sends *ok?* messages to each of its neighbors and changes back to the *wait_ok?* mode. The algorithm starts up with each agent sending *ok?* messages and going into the *wait_ok?* mode.

A quasi-local-minimum is a weaker condition than what is typically thought of as a local minimum. For an agent to be in a QLM they need to have at least one violated constraint, have a possible improvement of 0 and have each of their neighbors report a 0 improvement as well. Although, a QLM sometimes reports a minimum when one doesn't truly exist, it never fails to report one.

Because of the strict locking mechanism employed in the algorithm, the overall behavior of the agents is to simultaneously switch back and forth between the two modes. If one or more of the agents reacts slowly or messages are delayed, the neighboring agents wait for the correct message to arrive. This also makes the protocol's communication usage very predictable because in each mode, each agent sends exactly one message to each of its neighbors. Meaning that if there are m constraints, exactly $2m$ messages are transmitted during each step.

Overall, DBA can be classified as a two-step algorithm because it takes a sequence of two messages for any one agent to change its value. As will be shown, this makes the algorithm very reactive to changes in the environment. Along with that, the hill-climbing nature of the protocol provides fairly good results on problems that are unsatisfiable, but causes it to be unable to determine that the problem cannot be solved.

3.2 Modifying DBA for Dynamic Environments

Adapting the DBA protocol to work in dynamic environments is a fairly straight forward process. Figure 3 gives two additional procedures that are needed in the protocol. Both of these procedures can be triggered by an external event or by some internal change within the agents.

The *add constraint* procedure works by adding each new neighbor into a list of pending additions, then sends either an *ok?* or *improve?* message depending on the agent's mode. The pending additions list in necessary because of the strict locking mechanism within the protocol. Consider the following example. Let's say that agent x_i has had no neighbors for a while and is therefore in the *wait_ok?* mode, then is given a new neighbor x_j. Because it is in this mode, it will wait for x_j to send it a *ok?* message before continuing. For the sake of argument, let's say that x_j, who has had neighbors all along, is in the *wait_improve?* mode. It is expecting to get an *improve?* message from x_i. Since x_i is unaware of x_j's mode, it sends an *ok?* message assuming that x_j is in the same mode. Likewise, x_j sends an *improve?* message. When x_i and x_j receive their messages, they queue them because they are not in the proper mode, and the the protocol immediately deadlocks.

To avoid deadlocking like this, agents remain on the pending list until a message is received from them and the agent is in the correct mode to process it. So, in our example above, x_j can continue to process until it changes mode, then it processes the *ok?* message from x_i, moves x_i from pending to its neighbors list and sends x_i an *ok?* message. This causes x_i to synchronize with x_j and the protocol can continue executing.

The *remove constraint* procedure works by removing any neighbors that are unique to the removed constraint from the list of neighbors and from the pending list. It then deletes any stored *improve?* or *ok?* that may be awaiting processing. Finally, if the agent no longer has any neighbors, it changes to the *wait_ok?* mode. This last step prevents two agents that are not connected to any neighbors, then have a constraint added between them from deadlocking. To understand this point, consider the case when both of the

```
when add constraint (R_i) do
   add all new neighbors in R_i to pending;
   if mode == wait_improve
      send (improve, x_i, my_improve, current_eval)
         to new neighbors;
   else
      send (ok?, x_i, current_value) to new neighbors;
   end if;
end do;

when remove constraint (R_i) do
   remove all removed neighbors in R_i from neighbors;
   remove all removed neighbors in R_i from pending;
   if mode == wait_improve
      delete any stored improve messages from
            removed neighbors;
   else
      delete all removed neighbors from agent_view;
   if x_i has no neighbors
      mode == wait_ok;
end do;
```

Figure 3: The procedures for adding and removing constraints in Distributed Breakout.

agents are in different modes when the constraint is added between them.

4. DYNAMIC ASYNCHRONOUS PARTIAL OVERLAY

4.1 Asynchronous Partial Overlay

The Asynchronous Partial Overlay (APO) [6] algorithm is based on the concept of *cooperative mediation*. Cooperative mediation entails three main principles. The first is that agents should use local, centralized search to solve portions of the overall problem. Second, agents should use experience to dynamically increase their understanding of their role in the overall problem. Lastly, agents should overlap the knowledge that they have to promote coherence. Together these three ideas create a powerful paradigm which has been applied to several distributed problems. [5, 7]

Figures 4, 5, 6, 7, and 8 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, values, domains, and constraints of variables to which an agent is linked. The *good_list* 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 *good_list* 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 until one of the agents finds an unsatisfiable subsystem, or all of the conflicts have been removed.

```
procedure initialize
   d_i ← random d ∈ D_i;
   p_i ← sizeof(neighbors) + 1;
   m_i ← true;
   mediate ← false;
   add x_i to the good_list;
   send (init, (x_i, p_i, d_i, m_i, D_i, C_i)) to neighbors;
   initList ← neighbors;
end initialize;

when received (init, (x_j, p_j, d_j, m_j, D_j, C_j)) do
   Add (x_j, p_j, d_j, m_j, D_j, C_j) to agent_view;
   if x_j is a neighbor of some x_k ∈ good_list do
      add x_j to the good_list;
      add all x_l ∈ agent_view ∧ x_l ∉ good_list
         that can now be connected to the good_list;
      p_i ← sizeof(good_list);
   end if;
   if x_j ∉ 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_agent_view;
end do;
```

Figure 4: The APO procedures for initialization and linking.

4.1.1 Initialization (Figure 4)

On startup, the agents are provided with the value (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 its neighbors.

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 variable is only added to the *good_list* if it is a neighbor of another variable already in the list which ensures that the graph created by the variables in the *good_list* always remains connected.

4.1.2 Checking the agent view (Figure 5)

Whenever the agent receives a message which indicates a possible change to the status of its variable, it executes the check_agent_view procedure. In this procedure, the current *agent_view* (which contains the assigned, known variable values) is checked to identify conflicts between the variable owned by the agent and its **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.

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.

4.1.3 Mediation (Figures 6, 7, and 8)

The most complex and certainly most interesting part of the protocol is the mediation. 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 two-

```
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 ≠ ∅ or mediate ≠false do
      return;
   m'_i ← hasConflict(x_i);
   if m'_i and ¬∃_j(p_j > p_i ∧ m_j == true)
      if ∃(d'_i ∈ D_i) (d'_i ∪ agent_view does not conflict)
            and conflicts exclusively
            with lower priority neighbors
         d_i ← d'_i;
         send (ok?, (x_i, p_i, d_i, m_i)) to all x_j ∈ agent_view;
      else
         do mediate;
   else if m_i ≠ m'_i
      m_i ← m'_i;
      send (ok?, (x_i, p_i, d_i, m_i)) to all x_j ∈ agent_view;
   end if;
end check_agent_view;
```

Figure 5: The procedures for doing local resolution, updating the *agent_view* and the *good_list*.

```
procedure mediate
   preferences ← ∅;
   counter ← 0;
   for each x_j ∈ good_list do
      send (evaluate?, (x_i, p_i)) to x_j;
      counter ++;
   end do;
   mediate ← true;
end mediate;

when receive (wait!, (x_j, p_j)) do
   update agent_view with (x_j, p_j);
   counter --;
   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 agent_view with (x_j, p_j);
   counter --;
   if counter == 0 do choose_solution;
end do;
```

Figure 6: The procedures for mediating a session.

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, prevents the agent from engaging in two 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 actually link with those agents.

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 that value which is returned in an *evaluate!* message.

When the mediator has received either a *wait!* or *evaluate!* message from all of the agents that it has sent a request to, it computes a solution using a Branch and Bound search [4]. The goal of the search is to find a solution where all of the constraints are satisfied and the number of outside conflicts is minimized (like the min-conflict heuristic [8]).

If no satisfying assignments are found, the agent announces that the problem is unsatisfiable and the algorithm terminates. If a solution is found, *accept!* messages are sent to the agents in the session, *ok?* messages to the agents that are in its *agent_view*, but for whatever reason were not in the session, and the mediator sends *init* messages to any agent that is outside of its *agent_view*, but it caused conflict for as a result of selecting its solution.

Overall, APO has a very hill-climbing like nature when it first starts up because agents are only linked with their neighbors. As time goes on and the agents begin to centralize more of the problem, the problem solving become less parallel and more focused. However, since the initial hill climbing removes most of the constraint violations, massive parallelism is no longer needed. Based on the type and difficulty of the constraint violations, APO can act as either a one or four step protocol. Most often because of the mediation process, however, it takes four steps to change a variable's value.

4.2 Modifying APO for Dynamic Environments

Because we wanted to maintain its completeness, modifying the APO algorithm to work in dynamic environments was quite a bit more difficult than the modifications that were needed for DBA. Although APO's session based design lends itself to being able to handle changing constraints while it is in a mediation session, the biggest problem in the adaptation was in preventing the agents from continuously increasing the size of the problem they were centralizing. This necessitated the addition of a *remove* message.

The *remove* message is sent from one agent to another whenever they wish to remove their link. The main difficulty with the unlink request is that APO was designed to have bi-directional links. This means that if agent x_i is linked to x_j then x_j is linked to x_i. In fact, the completeness of the algorithm is dependent on this property because if agent x_i wants to mediate over x_j and x_i is high enough priority, then x_j must know about it and eventually allow the mediation to occur. This is a key point, because the bi-directionality of links in APO really means that if agent x_i has x_j in its *good_list* then x_j must have x_i in its *agent_view*.

This key insight means that agents can send requests for a removal when they want to take a variable out of their *good_list* but, may only remove the variable from their *agent_view* when they do not have it in their *good_list* and are sure that the owner of the variable does not have their variable in its *good_list*.

The mechanism used to handle removals involves maintaining two lists; one for remove requests sent and one for remove requests received. If an agent sends a remove request, it adds the variable to the removes sent list. Once it receives a response to the remove, it deletes the agent from

```
procedure choose_solution
   select a solution s using a search that:
        1. satisfies the constraints between agents in
           the good_list
        2. minimizes the violations for agents outside of
           the session
   if ¬∃s that satisfies the constraints do
      broadcast no solution;
   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, p_i, d_i, m_i, D_i, C_i)) to x_k;
            add x_k to initList;
         end if;
         send (accept!, (d'_j, x_i, p_i, d_i, m_i)) to x_j;
         update agent_view for x_j
      else
         send (ok?, (x_i, p_i, d_i, m_i)) to x_j;
      end if;
   end do;
   mediate ← false;
   check_agent_view;
end choose_solution;
```

Figure 7: The procedure for choosing a solution during an APO mediation.

its *agent_view* and its sent list. If an agent receives a remove request, it either immediately removes the variable from it *agent_view* and sends a response or it stores the request in its removes received list. If at some later time it wants to remove the variable, it removes the request from the removes received list and responses with a *remove*. If an agent sends a request or responds to a removal and later determines that it wants to relink with that variable, it must re-initialize the link by sending an *init*. This step ensures that the links and the remove lists remain correctly balanced.

In the current implementation of DynAPO, agents remove all non-neighbor variables from their *good_list* whenever a constraint is removed from their variable. The side effect of this is that in more dynamic environments, the agents very rarely extend their view past their immediate neighbors, making the algorithm behave like a four step version of DBA. As will be shown in the next section, the combination of limiting the view in this way along with the four step process of a mediation session causes the performance of DynAPO to degrade rapidly as the speed at which the constraints change increases.

Several modifications were also made to DynAPO to handle the times when the problem became unsatisfiable. The first change is that whenever a mediator identifies a subproblem that is unsatisfiable, it chooses a value for its variable that minimizes the conflicts with its neighbors. In addition, the mediator also announces that it has found an unsolvable subproblem by returning the value **null** in the *accept!* message that is sent to the agents at the end of the session. By sending the null value, the agents within the session become aware that mediator no longer wishes to work on the problem and that any one of them is free to take over as the mediator. Basically, this allows the lower priority agents to work on their subproblems in order to further minimize conflict.

Once an agent has discovered an unsatisfiable subproblem, they switch into a locally minimizing mode. During this

```
when received (evaluate?, (x_j, p_j)) do
   m_j ← true;
   if mediate == true or ∃k(p_k > p_j ∧ m_k == true) do
      send (wait!, (x_i, p_i));
   else
      mediate ← 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, p_i, labeled D_i));
   end if;
end do;

when received (accept!, (d, x_j, p_j, d_j, m_j)) do
   d_i ← d;
   mediate ← false;
   send (ok?, (x_i, p_i, d_i, m_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 8: Procedures for receiving a session.

mode, whenever an agent checks its *agent_view*, it calculates the minimal conflicting value for its variable and if it is not in or expecting a mediator session, changes to the new value. Agents move out of this mode is one of their constraints is removed, or if one of the constraints on the agents in their *good_list* is removed. This ensures that the agents always continue to work when it is possible that a solution might exist and they haven't found it.

5. EXPERIMENTAL SETUP AND RESULTS

To test the DynDBA and DynAPO algorithms we used a modified version of the distributed 3-coloring problem. In distributed 3-coloring, each agent is assigned one variable and given the names of their neighbors in the constraint graph. The goal of the agents is to find a coloring for its variable such that it is different from the colors of its neighbors. The domain of each of the variable is, as the name implies, the same three colors.

In the dynamic version, a simulator is used to modify the constraints on the variables as the scenario unfolds. During these experiments, the simulator changed the constraints at a rate of δ constraints per execution cycle. So for instance, at a rate of $\delta = 3$, three constraints are added or removed each cycle. To ensure that the problem remained at a constant difficulty, or overall graph density, the number of adds and removes per unit time were always equal. In essence, the constraints were changed. For the sake of clarity, during a cycle, each agent is given control, allowed to process each of its incoming messages and queues up outgoing messages to be sent at the beginning of the next cycle. The amount of actual processing time allocated to each agent varies depending on the amount of work needed to process the messages, compute solutions, etc.

For the test series, 30 node graphs were used. The test series varied the edge density of the graphs, testing graphs with average degrees of 2.0, 2.3, and 2.5. These particular values were chosen because they represent three distinct region in the phase transition for random 3-coloring graphs [1]. Graphs with a density of 2.0 are likely to be satisfiable, 2.3 are within the phase transition, and 2.5 are likely to be unsatisfiable.

Density	δ	DynAPO Mean			DynDBA Mean		
		Error	Messages	% Correct	Error	Messages	% Correct
2.0	0.2	0.9	28,035	50.6	1.6	119,958	25.7
	1.0	2.1	44,507	16.6	2.6	120,758	6.7
	2.0	3.2	65,306	5.7	3.4	121,754	2.7
	4.0	4.9	96,317	1.2	4.5	123,519	0.8
	6.0	6.2	123,939	0.3	5.5	125,277	0.3
2.3	0.2	1.9	62,285	26.0	2.2	137,922	15.2
	1.0	3.1	73,407	8.9	3.5	138,722	4.0
	2.0	4.3	90,491	2.8	4.1	139,718	1.9
	4.0	6.2	121,717	0.6	5.3	141,429	0.5
	6.0	7.6	147,730	0.2	6.3	143,147	0.2
2.5	0.2	2.8	91,673	12.2	2.8	149,898	8.6
	1.0	4.0	95,352	5.5	4.0	150,698	2.7
	2.0	5.1	109,916	1.5	4.6	151,694	1.2
	4.0	6.9	138,353	0.4	5.7	153,430	0.4
	6.0	8.4	164,298	0.2	7.6	155,073	0.2

Figure 9: Results of testing DynAPO and DynDBA on 30 node, dynamic 3-coloring problems.

The series also varied the rate of constraint changes in the environment. Values of $\delta = (0.2, 1.0, 2.0, 4.0, 6.0)$ were tested where the number of constraint additions and removals were kept in balance. For each combination of graph density and δ, 20 graphs were generated, then the simulation run for 1000 cycles. So, for $\delta = 6.0$, 3 constraint were added and 3 were removed each cycle making a total of 6000 changes to the original graph over the run. To make the comparison fair between the two methods, the seeds used for the random number generator were saved and reused. This ensured that each algorithm was exposed to exactly the same set of graphs, initial starting values, and the same series of changes.

During the run, data was collected on the number of constraint violations at each time step and the number of messages used. In addition, whenever one or more constraints were added or removed from the previous step, the optimal bound was computed. This allow for the error and "correctness" of each algorithm to be measured. The error is the mean of the difference between the optimal value and the current solution for each cycle. The correctness is calculated as the percentage of time that the algorithm's solution satisfied the optimal number of constraints.

The results of these tests can be seen in Table 9 and figure 10, 11, and 12. As can be seen, both protocols significantly outperform a static solution, which is a solution to the initial problem that does not change over time. In addition, DynAPO had a smaller mean error on problems that changed slowly. As the rate of change increased, however, the two-step nature of the DynDBA protocol began to dominate. These results are actually fairly intuitive once one considers the number of constraint changes that take place during a single DynAPO mediation session.

The next interesting result from these tests is that DynAPO used considerably fewer messages than DynDBA on slowly changing problems, but its usage increased rapidly as δ increased. The most reasonable explanation for this behavior is that when the problem changes slowly, DynAPO often reaches quiescence between successive changes. In addition, because constraint removals happen very infrequently, the *agent_view*s of the agents grow quite large. This means that when the agents mediate, change values, add or remove constraints they have to send a large number of messages to inform all of the agents that they are link to about the

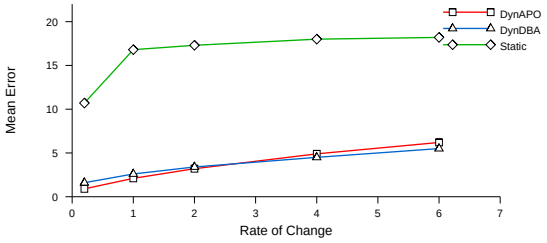

Figure 10: Instantaneous error of DynAPO, DynDBA, and a static solution on 3-coloring instances with density = 2.0n.

change. As δ increases, the agents communicate more frequently to keep their internal states up-to-date. In addition, because constraints are removed very frequently, the agents maintain very small subproblems and therefore only mediate with a small number of agents decreasing the standard deviation.

The last result that bares mentioning is the percentage of time that each of the protocols has a correct solution. As the result show, DynAPO is, on average, more accurate than DynDBA. There are a number of explanations for this, but the most likely cause is that DynAPO is a complete method and DynDBA is an incomplete method. This means that when no solution exists, DynAPO is aware of it, and just stops working. DynDBA on the other hand, keeps trying to solve the problem which causing it to move away from the optimal solution.

6. CONCLUSIONS AND FUTURE WORK

This paper presented two new algorithms for solving dynamic, distributed constraint satisfaction problems. The first algorithm is called DynDBA and is an adaptation of the DBA algorithm. The second, is an adaptation of the APO algorithm and is called DynAPO. A new formulation of the Dynamic DCSP was also presented which directly relates the amount of change in the DCSPs to time. Using this formulation an algorithms performance can be measured in relation to how quickly the problem changes during the problem solving process. Lastly, the DynAPO and DynDBA algorithms were tested using a dynamic version of the dis-

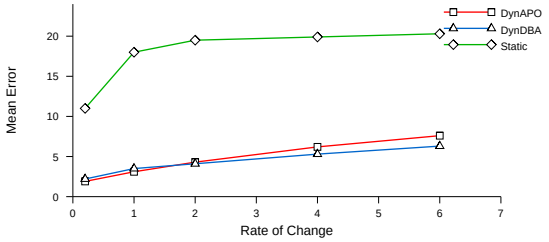

Figure 11: Instantaneous error of DynAPO, DynDBA, and a static solution on 3-coloring instances with density = 2.3n.

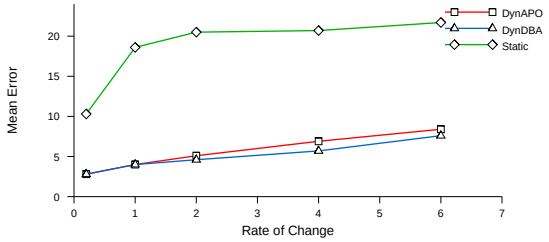

Figure 12: Instantaneous error of DynAPO, DynDBA, and a static solution on 3-coloring instances with density = 2.5n.

tributed 3-coloring problem. The results of these tests were mixed, with neither algorithm being completely dominate. This seems to indicate that determining which algorithm is best largely depends on the dynamics of the environment, the problems overall difficulty, and a users particular needs.

There is a vast amount of future work to be done. First and foremost, additional algorithms need to be tested and additional dynamic environments need to be developed. It would be interesting to see how a one-step algorithms like P2P performs and whether something more general could be found about the relationship between the rate of change in a problem and the best problem solving strategy. In addition, it seems, at this point, clear that algorithms could be developed which monitor environmental dynamics and alter their behavior to optimize their performance.

Lastly, although the simulators used for these tests allowed us to relate a notion of time to change, it did not relate real-time to change. One has to wonder about the exact performance of these algorithms when the cost for computation during a cycle becomes explicit.

7. REFERENCES

[1] J. Culberson and I. Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 265(1–2):227–264, 2001.

[2] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, 1988.

[3] S. Fitzpatrick and L. Meertens. *Distributed Sensor Networks: A Multiagent Perspective*, chapter Distributed Coordination Through Anarchic Optimization, pages 257–294. Kluwer Academic Publishers, 2003.

[4] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1–3):21–70, 1992.

[5] R. Mailler and V. Lesser. Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 438–445. IEEE Computer Society, 2004.

[6] R. Mailler and V. Lesser. Using Cooperative Mediation to Solve Distributed Constraint Satisfaction Problems. In *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, volume 1, pages 446–453, New York, 2004. IEEE Computer Society.

[7] R. Mailler, R. Vincent, V. Lesser, J. Shen, and T. Middlekoop. Soft real-time, cooperative negotiation for distributed resource allocation. In *Procceedings of the 2001 AAAI Fall Symposium on Negotiation*, 2001.

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

[9] P. J. Modi, H. Jung, M. Tambe, W.-M. Shen, and S. Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In J.-J. Meyer and M. Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 181–193, 2001.

[10] P. Morris. The breakout method for escaping local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 40–45, 1993.

[11] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic csp. In *Proceedings pf the Fifth IEEE International Conference on Tools with Artificial Intelligence*, 1993.

[12] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[13] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 307–312, 1994.

[14] M. 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.

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