Solving Distributed CSPs using Dynamic, Partial Centralization without Explicit Constraint Passing [*]

Roger Mailler
SRI International
333 Ravenswood Dr
Menlo Park, CA 94025
mailler@ai.sri.com

ABSTRACT

Dynamic, partial centralization is a technique which has recently begun to receive a considerable amount of attention in the distributed problem solving community. As the name implies, this technique works by dynamically identifying portions of a shared problem to centralize in order to speed the problem solving process. Currently, a number of algorithms have been created which employ this simple, yet powerful technique to solve problems such as distributed constraint satisfaction (DCSP), distributed constraint optimization (DCOP) and distributed resource allocation.

In fact, one such algorithm, Asynchronous Partial Overlay (APO), was recently shown to outperform the Asynchronous Weak Commitment (AWC) protocol which is one of the best known methods for solving DCSPs. One of the key differences in these two algorithms is that APO, as part of the centralization process, uses explicit constraint passing which AWC assumes, for reasons of security, privacy, or shear size, cannot be done.

This paper introduces a new algorithm which is a hybrid of AWC and APO that like AWC uses nogood passing to provide security and privacy and like APO uses dynamic partial centralization to speed the problem solving process. Like its parents, this new algorithm, called APO-Nogood, is sound and complete and as will be shown, performs nearly as well as APO, while still outperforming AWC, on distributed 3-coloring problems.

1. INTRODUCTION

Over the years, distributed problem solving has received a great deal of attention for a number of reasons. Probably the two most compelling reasons are the expected speed up associated with using more than a single processor and the need for the problem solvers to maintain some degree of privacy and security. These two goals can often be quite contradictory because, intuitively, the more information an agent is willing to reveal upfront as part of the problem solving process, the faster a solution can be computed.

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

Recently a new methodology for solving distributed problems, called dynamic, partial centralization, has come into focus and is beginning to recent a fair amount of attention. This technique can best be described as a focused, incremental and asynchronous centralization of portions of a shared problem in order to rapidly converge on solutions. Several protocols have already been created that use this hybrid centralized/distributed search technique and have been shown to outperform existing distributed protocols.

One of the key characteristics of each of these algorithms, however, is that the agents have to be willing to reveal a great deal of information to other agents whenever it is requested. For example, in Asynchronous Partial Overlay (APO) [7], agents willingly reveal their constraints and the possible values of their variable whenever they are directly asked. Contrast this with the Asynchronous Weak Commitment (AWC) protocol [10] where agents only reveal information about their constraints and possible values when they reach a *deadend* in the problem solving process by created a *nogood*. The willingness to reveal information is one of the reasons, although not the only one, that APO outperforms AWC across a wide spectrum of problem sizes and difficulties.

The purpose of this paper is to present a new algorithm, called APO-Nogood, which is a hybrid of AWC and APO. Like APO, this new algorithm uses dynamic, partial centralization and like AWC only reveals information, in the form of a nogood, when necessary during the problem solving process. The two main goals in creating this algorithm are to show that although partial centralization involves revealing knowledge in order to solve a shard problem, the knowledge that is exposed can be minimized, obscured, or revealed in an incremental manner. The second goal is to demonstrate that even when constraints are not explicitly revealed, that dynamic, partial centralization still outperforms the AWC-like trial-and-error approach to solving distributed problems.

The rest of this paper is organized as follows. In the next section, we will introduce the distributed constraint satisfaction problem. We will go on to describe the APO-Nogood algorithm, give an example of its execution on a simple problem, and mention the issues of soundness and completeness. We will then present the setup for our experimental evaluation followed by the results. Finally, we will present our conclusions for this work.

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 $R_i(d_{i1}, \ldots, d_{ij})$ is a predicate on the Cartesian product $D_{i1} \times \cdots \times D_{ij}$ 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. 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.

In this paper 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. It is fairly easy to extend our approach to handle the case where there are more than one variable per agent. We also often use the term *constraint graph* to mean the graph created by representing the variables as nodes and the constraints as edges. In this way, a variable can also be referred to as a node (in the constraint graph). As a strictly definitional point, two variables are considered to be *neighbors* if they share a constraint.

3. APO-NOGOOD

3.1 Asynchronous Partial Overlay (APO)

Due to space limitation, the entire APO algorithm cannot be detailed in this paper. We, instead, refer the reader to [7]. The basic layout of the APO protocol, however, is presented in figure 1.

Conceptually, APO is based on the *cooperative mediation* paradigm. 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. [6, 8]

The APO algorithm works by constructing two main data structures; the *good_list* and 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

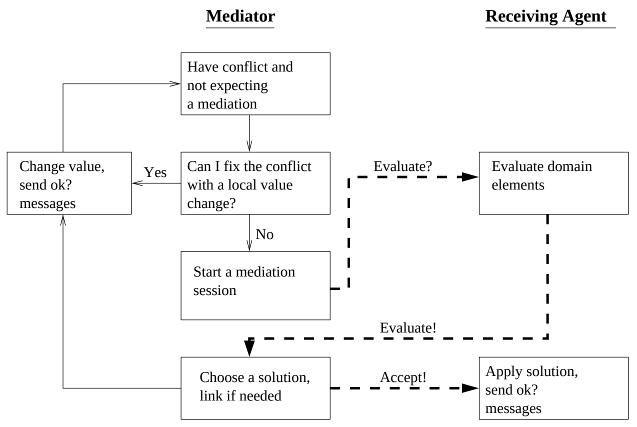
Mediator
Receiving Agent
Have conflict and not expecting a mediation
Change value, send ok? messages
Yes
Can I fix the conflict with a local value change?
Evaluate?
Evaluate domain elements
No
Start a mediation session
Evaluate!
Choose a solution, link if needed
Accept!
Apply solution, send ok? messages

Figure 1: The basic APO protocol.

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, the agents try to solve the subproblem they have centralized within their *good_list* or determine that this subproblem is unsolvable (indicating that the entire problem is overconstrained). To do this, whenever an agent recognizes a constraint violation involving its variable, it takes the role of the mediator and attempts 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 which increases the size of the good_list. This process continues until one of the agents finds an unsatisfiable subsystem, or all of the conflicts have been removed.

Like AWC, agents that use APO have a dynamic priority value which is used to determine which agent mediates when a conflict is detected. Currently, the heuristic for setting this priority value is to use the size of the subproblem that the agent knows. Although one could conceive of a number of other heuristics which optimize different metrics, this particular heuristic was chosen to minimize the number of parallel cycles needed to compute a solution.

One of the most important steps in the APO protocol is the linking step that occurs during startup and at the end of a mediation session. The linking step ensures that, in the limit and only when necessary, one or more of the agents centralizes the entire problem in order to ensure the completeness of the search.

The linking step in APO entails having the agents exchange the possible domain values, D_i and the set of constraints, $\forall R_i \; x_i \in R_i$, on the agent's variable. In many domains, particular in ones where every agent is trusted and cooperative, the open exchange of this knowledge is quite acceptable and leads to significant improvements in the runtime of the algorithm. However, there are times when, for reasons of privacy, security, or sheer size, exchanging this information upfront is impossible or cost prohibitive.

3.2 APO-Nogood

The APO-Nogood algorithm is very similar in nature to the APO algorithm. The key difference is that instead of directly exchanging constraints, the agents exchange *nogoods* in a similar fashion to the AWC algorithm as part of the problem solving process. For clarity, a *nogood* is a set of variable/value pairs which either directly violates a constraint or leads to the violation of a constraint through an indirect relationship.

By exchanging nogoods, instead of constraints, the agents gain two things. The first is that the agents only reveal portions of their constraints incrementally, which means they may not have to reveal all of the details of their constraints in order to solve a problem. This is particular important in domains where the variables have very large domains. The second is that agents can obscure the actual nature of the constraints they have on their variable by padding the most minimal nogood with additional variable/value pairs. By padding them in this way, it is harder for another agent to actually know the details of the constraints, but slows the execution of the algorithm because it is harder to identify when the problem is unsolvable.

There are several secondary effects of changing the algorithm in this way. Probably the most important is that the agents need to maintain a *nogood list* which is used to remember the constraints the agent has identified during the problem solving process. Like AWC, the size of the nogood list can grow quite large (exponential in the worst case), especially if the agents try to hide their direct constraints by padding their nogoods. However, if the agents are willing to exchange nogoods that are directly derived from their constraints, the size of the nogood list becomes quite manageable being directly related to the number and complexity of the constraints as opposed to the number of possible assignments to the variables.

3.2.0.1 *Initialization.*

Like APO, on startup, the agents are provided with the value (they pick it randomly if one isn't assigned) and the constraints on their variable. Using these constraints, the agents derived the direct nogoods which are placed in their nogood lists. Unlike APO however, initialization proceeds by having each of the agents send out an "ok?" message to its neighbors. The content of this message is considerably different from the "ok?" messages in APO. In APO-Nogood, the agents send their current priority, the value of their variable, their variable's current domain, and the current set of violated nogoods from their nogood list that involve their variable.

Agents send their domain values as part of the "ok?" message because it ensures that the mediator always has the current set of allowable values for the variables in its good_list. This is particularly important if an agent calculates that one of its values is not arc-consistent. This can be thought of as the agent deriving a unary nogood which disallows one of its variable's values.

The "ok?" message also includes the set of currently violated nogoods that include the agent's variable. There are two reason for including this information. First, when this set is empty, it indicates that the agent does not wish to mediate. Second, as will be illustrated later, this information is used to ensure that mediators are informed of inadvertent nogood violations that result from changing the values of multiple variables in a session without knowing that they are related to one another.

When an agent receives an "ok?" message (either during the initialization, through a later link request, or as a state update), it records the information in its agent_view and adds the variable to the good_list if it can. A variable is only added to the good_list if it shares a nogood with a variable which is already in the list which ensures that the graph created by the variables in the good_list always remains connected.

3.2.0.2 *Checking the agent view.*

Whenever the agent receives a message which indicates a possible change to the status of its variable, it checks the current agent_view (which contains the assigned, known variable values) to identify violated nogoods that involve the variable owned by the agent. If, during this check, an agent finds a violation 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 violation(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 violations, 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's at a deadend), it starts a mediation session.

3.2.0.3 *Mediation.*

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 twofold. First, it informs the receiving agent that a mediation is about to begin and tries to obtain a lock from that agent. This lock 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.

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 an agent of higher priority than the requester. If the agent is available, it labels each of its domain elements with the nogoods that would be violated 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 [2]. The goal of the search is to find a solution where no nogoods are violated and the number of outside conflicts is minimized (like the min-conflict heuristic [9]). During this search, new nogoods can be derived using nogood learning [3]. These nogoods are recorded in the nogood list and can be used during subsequent searches to prune the search space.

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 and "ok?" messages are sent to the agents that are in its agent_view, but, for whatever reason, were not in the session, and to any agent that is not in its agent_view, but it caused conflict for as a result of selecting

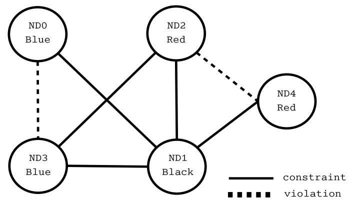

Figure 2: Example 3-coloring problem with 5 variables and 7 "not-equals" constraints.

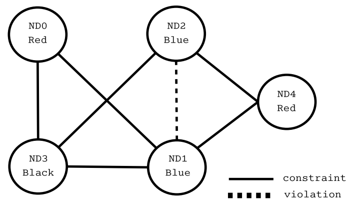

Figure 3: Sample problem after first mediation.

its solution.

3.3 Example Execution

Consider the 3-coloring problem in figure 2. In this problem there are 5 variables, each assigned to an agent and 7 constraints which represent the "not equals" predicate. Being a 3-coloring problem, the variables can only take the value black, blue, or red. There are currently two constraint violations, between ND2 and ND4 and between ND0 and ND3.

On initialization, each of the agents adds nogoods to their nogood lists for the constraints that they have on their variable. They then send "ok?" messages to the agents they share constraints with (their neighbors).

Once the initialization has complete each of the agents checks its *agent_view* to determine if its variable is involved in a violation. In this case, ND0, ND2, ND3, and ND4 determine that have a conflict. Because of the priority ordering, ND4 (priority 3) waits for ND2 (priority 4) to mediate. ND0 (priority 3) and ND2 wait for ND3 (priority 3 tie broken by name). ND3, knowing it is higher priority than ND0 and ND2, first checks to see if it can resolve its conflicts by changing its value, which it can't. It then starts a mediation session and sends "evaluate?" messages to ND0, ND1, and ND2.

Upon receiving the "evaluate?" messages, ND0, ND1, and ND2 evaluate their domain elements to identify the nogoods that would be violated by each of them. This information is then returned to ND3 in an "evaluate!" message. The following are the labeled domains for the agents in the session with ND3:

- ND0 - Black violates (ND0=Black,ND1=Black); Blue violates (ND0=Blue,ND3=Blue); Red causes no violations

- ND1 - Black cause no violations; Blue violates (ND1=Blue, ND0=Blue) and (ND1=Blue,ND3=Blue); Red violates (ND1=Red,ND2=Red) and (ND1=Red, ND4=Red)

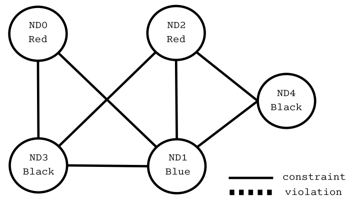

Figure 4: Sample problem after second mediation.

- ND2 - Black violates (ND2=Black,ND1=Black); Blue violates (ND2=Blue, ND3=Blue); Red violates (ND2=Red, ND4=Red)

Upon receiving these responses, ND3 computes a solution that changes the values of all of the variables in the session (see figure 3). Based on the information that ND3 obtained from the "evaluate!" messages, it believes that this solution solves its subproblem and causes no conflicts for agents outside of the session. ND3 sends "accept!" message to the agents in the session.

Upon receiving the "accept" messages, each agent changes its value and checks its agent_view. This time, ND1 and ND2 are in conflict which was caused by ND3 changing their values to blue which inadvertently caused the violation between them. To prevent this from happening again, the "ok?" messages that are sent by ND1 and ND2 include their current conflict set. This allows ND3 to learn of the relationship between ND2 and ND1 so it doesn't repeat the same error.

Since ND1 and ND2 have a conflict, ND1 is higher priority (priority 5), and cannot solve the conflict by making a local value change, it starts a mediation session. Below are the responses to the "evaluate?" messages sent by ND1:

- ND0 - Black violates (ND0=Black,ND3=Black); Blue violates (ND0=Blue,ND1=Blue); Red causes no violations

- ND2 - Black violates (ND2=Black,ND3=Black); Blue violates (ND2=Blue,ND1=Blue); Red violates (ND2=Red,N4=Red)

- ND3 - Black causes no violations; Blue violates (ND3=Blue, ND1=Blue)and (ND3=Blue,ND2=Blue); Red violates (ND3=Red,ND0=Red)

- ND4 - Black causes no violations; Blue violates (ND4=Blue, ND2=Blue)and (ND4=Blue,ND1=Blue); Red causes no violations

ND1 computes a solution which changes its value to black and ND2's to red and sends "accept!" messages. All of the agent's check their agent_view find no conflicts so the problem is solved (figure 4).

3.4 Soundness and Completeness

The soundness and completeness of the APO-Nogood algorithm are derived directly from the soundness and completeness of APO. We refer the reader to [5] for the complete details of the proofs for APO. Here is a basic outline of the proof for APO-Nogood:

Density	Cycles Mean			Messages Mean			PTime Mean		
	APO	AWC	NAPO	APO	AWC	NAPO	APO	AWC	NAPO
1.8	2,816	2,490	2,956	892	1,686	927	391	618	499
2.0	3,200	12,583	3,906	1,283	31,426	1,566	517	9,392	784
2.3	2,610	16,760	4,410	1,301	59,882	2,492	508	16,624	1,493
2.5	2,066	28,196	3,470	1,337	136,479	2,450	410	5,5826	1,477
2.7	1,570	23,023	3,113	1,157	122,305	2,350	391	36,501	1,487

Table 1: Results of testing APO, AWC, and APO-Nogood on 30 node, 3-coloring problems of various density.

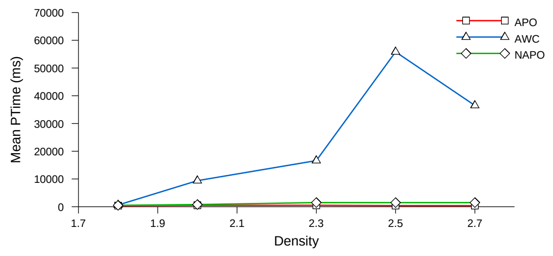

Figure 5: Average parallel cycle time (milliseconds) for APO, AWC, and APO-Nogood on 30 node, 3-coloring problems of various density.

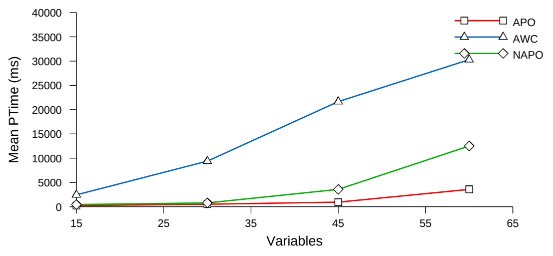

Figure 6: Average parallel cycle time (milliseconds) for APO, AWC, and APO-Nogood on 3-coloring problems with $density = 2.0n$.

- 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 nogood is violated, someone will try to fix it. The protocol is dead-lock free. The other half of the soundness.
- Eventually, 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 is done by collecting nogoods through "evaluate!" messages and "ok?" messages. This ensures completeness.

4. EXPERIMENTS

To test the APO-Nogood algorithm, we implemented the AWC, APO and APO-Nogood algorithms and conducted experiments in the distributed 3-coloring domain. The particular AWC algorithm we implemented can be found in [11] which includes the resolvent *nogood* learning mechanism described in [4]. We conducted 2 sets of experiments.

In the first set of experiments we compared the algorithms using 30 variable, randomly generated graph coloring problems while varying the edge densities across the known phase transition for 3-coloring problems [1]. During these tests we measured the number of messages, the number of *cycles*, and the parallel cycle times (denoted PTime in the graphs) used by the algorithms. During a 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. 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 parallel cycle time is calculated by summing, over each of the cycles, the maximum time used by any single agent during the cycle. In the second set of experiments, we tested the scalability of the algorithms by varying the size of the problems from 15 to 60 variables in the three major regions of the phase transition. Again, we collected the number of messages, cycles, and parallel cycle times for each of the algorithms. The values reported in these experiments are the mean for 30 instances for each data point where each algorithm was given the exact same instances for fairness.

The result of these experiments can be seen in tables 1 and 2 and figures 5 through 8. From these graphs and tables, it is easy to see that both APO and APO-Nogood outperform AWC on all of the metrics that were measured. In addition, APO outperforms APO-Nogood because as was mentioned earlier, it takes APO-Nogood additional time to collect enough information to solve a problem or prove it is unsolvable. One particularly interesting result is the scalability of the parallel cycle time of APO-Nogood (table 2). The difference between solving 45 and 60 node problems is very large. This is clearly caused by the time it takes the centralized solver to process the ever increasing number of nogoods that are in the agents' nogood lists. If a better nogood based solver had used, these values would be considerably smaller than the ones reported here.

5. CONCLUSION

In this paper, we presented a new hybrid AWC/APO algorithm called APO-Nogood. As was shown in experimentation, this algorithm, like APO, outperform AWC on a 3-coloring problems of various size and density on several metrics. By creating this algorithm, we showed that constraint passing is not necessary in an algorithm that is based on dynamic, partial centralization and that the likely reason why algorithms like APO outperform AWC is the combination of distributed/centralized search techniques they use.

N	D	Cycles Mean			Messages Mean			PTime Mean		
		APO	AWC	NAPO	APO	AWC	NAPO	APO	AWC	NAPO
15	2.0	1,666	6,646	2,840	346	5,389	656	235	2,449	446
	2.3	656	7,490	1,720	192	8,242	534	105	3,380	381
	2.7	356	7,213	1,306	119	9,701	491	58	2,693	322
30	2.0	3,200	12,583	3,906	1,283	31,426	1,566	517	9,392	784
	2.3	2,610	16,760	4,410	1,301	59,882	2,492	508	16,624	1,493
	2.7	1,570	23,023	3,113	1,157	122,305	2,350	391	36,501	1,487
45	2.0	5,216	15,333	6,600	2,937	71,127	3,828	942	21,654	3,577
	2.3	8,366	57,393	8,533	7,895	526,501	7,427	2,337	155,001	12,830
	2.7	2,053	38,672	3,433	2,391	455,133	4,158	602	101,187	10,083
60	2.0	9,316	17,683	11,083	7,354	110,064	7,294	3,574	30,293	12,511
	2.3	12,570	62,280	14,400	15,804	787,424	18,630	10,140	456,294	148,321
	2.7	3,016	57,285	3,938	4,798	1,190,104	6,792	2,702	448,429	118,520

Table 2: Results of testing APO, AWC, and APO-Nogood on 3-coloring problems of various size and density.

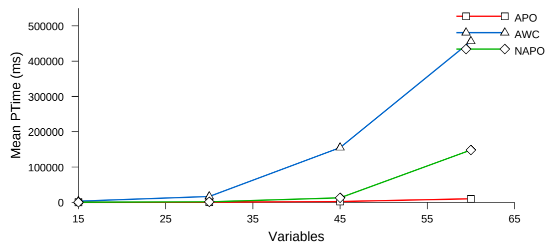

Figure 7: Average parallel cycle time (milliseconds) for APO, AWC, and APO-Nogood on 3-coloring problems with $density = 2.3n$.

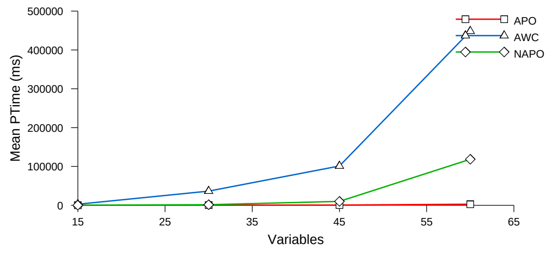

Figure 8: Average parallel cycle time (milliseconds) for APO, AWC, and APO-Nogood on 3-coloring problems with $density = 2.7n$.

6. REFERENCES

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

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

[3] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of the Twelfth Natioanl Conference on Artificial Intelligence*, pages 294–300, 1994.

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

[5] R. Mailler. *A Mediation-Based Approach to Cooperative, Distributed Problem Solving*. PhD thesis, University of Massachusetts, Amherst, MA, 2004.

[6] R. Mailler and V. Lesser. Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.

[7] R. Mailler and V. Lesser. Using Cooperative Mediation to Solve Distributed Constraint Satisfaction Problems. *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.

[8] 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.

[9] 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.

[10] 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.

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