Improving the Privacy of the Asynchronous Partial Overlay Protocol

Roger Mailler

University of Tulsa

Department of Computer Science

Tulsa, OK

mailler@utulsa.edu

(918)-631-3140 (phone)

(918)-631-3077 (fax)

February 29, 2012

Abstract

Dynamic, partial centralization has received 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 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 between these two algorithms is that APO, as part of the centralization process, uses explicit constraint passing. AWC, on the other hand, passed *nogoods* because it tries to provide security and privacy. Because of these differences in underlying assumptions, a number of researchers have criticized the comparison between these two protocols.

This article attempts to resolve this disparity by introducing a new AWC/APO algorithm called Nogood-APO that like AWC uses nogood passing to provide privacy and like APO uses dynamic, partial centralization to speed the problem solving process. Like its parent algorithms, this new protocol is sound and complete and performs nearly as well as APO, while still outperforming AWC, on distributed 3-coloring problems. In addition, this paper shows that Nogood-APO provides more privacy to the agents than both APO and AWC on all but the sparsest problems. These findings demonstrate that a dynamic, partial centralization-based protocol can provide privacy and that even when operating with the same assumptions as AWC still solves problems in fewer cycles using less computation and communication.

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

1 Introduction

Over the years, distributed problem solving has received a great deal of attention for a number of reasons. The most compelling are that some problems are naturally distributed, multiple processor can compute solutions faster, and privacy and security can be maintained. These reasons can often be quite contradictory because, for example, the more information an agent is willing to reveal upfront as part of the problem solving process, the faster a solution can be computed.

One methodology for solving distributed problems, called dynamic, partial centralization tries to solve naturally distributed problems in the fastest manner possible by performing focused, incremental and asynchronous centralization of portions of a shared problem. Several protocols have already been created that use this hybrid centralized/distributed search technique and have been shown to outperform existing protocols on a large number of distributed problems.

One of the key characteristics of each of these algorithms, however, is that the agents have to be willing to directly reveal a great deal of information to each other. For example, in Asynchronous Partial Overlay (APO) [13], agents reveal their variable's constraints and domain whenever requested. The Asynchronous Weak Commitment (AWC) protocol [28], on the other hand, only reveals information about a variable's constraints and domain, using a *nogood*, when it reaches a *deadend* in the problem solving process. The willingness to reveal information is one reason, although not the only one, that APO outperforms AWC across a wide spectrum of problem sizes and difficulties.

The purpose of this article is to present a new hybrid AWC/APO algorithm, called Nogood-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 shared 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 article is organized as follows. The next section introduces the distributed constraint satisfaction problem and provides a review of the techniques used to solve them. It goes on to describe the Nogood-APO algorithm, give an example of its execution on a simple problem, and mention the issues of soundness and completeness. This article then presents the experimental evaluation followed by the results. The final section presents conclusions.

2 Background

2.1 Distributed Constraint Satisfaction

A Distributed Constraint Satisfaction Problem (DCSP),
$P = \langle V, A, D, R \rangle$, consists of the following [29]:

- A set of n variables $V = \{x_1, \ldots, x_n\}$.
- A set of g agents $A = \{a_1, \ldots, a_g\}$
- Discrete, finite domains for each of the variables $D = \{D_1, \ldots, D_n\}$.

- A set of m 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 $S = \{d_1, \ldots, d_n | d_i \in D_i\}$ such that each of the constraints in R is satisfied. DCSP, like its centralized counterpart CSP, has been shown to be NP-complete, making it unlikely that a solution can be found without conducting some form of search.

In this work, we focus on the case where each agent is assigned a single variable and the constraints are binary. Since each agent is assigned a single variable, we will refer to the agent by the name of the variable it manages. Because the constraints are binary, we can refer to the graph created by representing variables as vertices and constraints as edges as the *constraint graph*. In addition, two variables are considered to be *neighbors* if they share a constraint.

2.2 Protocols for DCSP

Like their centralized counterparts, the set of algorithms that are used to solve DCSP problems can be broken down into two major categories: incomplete and complete. Both of these techniques have their advantages and disadvantages as we will discuss in the following sections.

2.2.1 Incomplete Methods

The family of the incomplete distributed algorithms are based on centralized hill-climbing methods like GSAT and WalkSAT [26]. Although in practice these algorithms perform well, they do not guarantee that a solution will be found when it exists and cannot identify when there is no solution. This

is because these algorithms are prone to getting trapped in local minima. Accordingly, various heuristics have been devised to escape local minima.

The **Distributed Breakout Algorithm (DBA)** [31] is a distributed adaptation of the centralized Breakout algorithm [23]. DBA works by alternating between two modes. In the first mode, called the *wait_ok?* mode, the agents wait until they have received an *ok?* messages from each of their neighbors. These messages allow the agents to accurately calculate the cost of their local variable. Once the agents have a clear picture of the state of their variable, they calculate the best alternative value assignment and the associated improvement to the cost function. The agents then sends out an *improve?* message, which contains their improve value, to each of their neighbors and change to the *wait_improve?* mode. In the *wait_improve* mode, the agents collect *improve?* messages from each of their neighbors. Once all of the messages have been received, the agents check to see if their improvement is the best among their neighbors. If it is, it changes its value to the new improved value. If an agent believes it is in a local minima, it increases the weights on all of its suboptimal cost functions. Finally, the agents send *ok?* messages to each of their neighbors and change back to the *wait_ok?* mode. The algorithm starts up with each agent sending *ok?* messages and going into the *wait_ok?* mode. This two phase synchronizing mechanism was devised to prevent collisions caused by neighbors simultaneously changing values. However, it only allows agents to change value every other phase and restricts the number of variables that can change to less than half in a fully connected problem. The DBA-WP and DBA-SP protocols are two variants of the DBA protocol have been created to improve its overall parallelism and prevent pathological behavior by introducing randomness [35].

The **Distributed Stochastic Algorithm (DSA)** is one of a class of

algorithms that changes each variable to its best value with some probability $p \in [0, 1]$. As each variable changes with probability p, the likelihood of two neighbors having collision is p^2. If p is chosen carefully, the protocol will hill climb to a better state. One of the greatest benefits of the DSA protocol is that it uses considerably fewer messages than the Distributed Breakout Algorithm [31] because agents communicate only when they change values. As the protocol executes and the number of violations decrease, so do the number of messages. DSA has a number of implementation variants. Zhang, et.al., [34, 33] experimented with changing an agent's value v with probability p at times when there is improve (an expected hill-climbing move) and when there is no improve with and without a conflict existing in the world (a lateral move). However, the same value of p was used for all cases. Their work focused on the effects of an agent having the chance to change in different variations of those situations, and found that one variant, DSA-B, was the best approach because it is able to escape certain types of local minima in the search space by making lateral moves.

The **Distributed Probabilistic Protocol (DPP)** builds on ideas from both the DSA and DBA protocols [16]. Like DSA, it uses probabilities to determine when the agents should take action. It is like DBA in that it exchanges both *ok?* and *improve* messages. Because the DPP protocol uses a dynamic mixture of randomness and direct control, the probability of an agent taking an action is strongly associated with the structure and current state of the problem. DPP works by having agents exchange probability distributions functions (PDFs) that describe the likelihood they are going to have a particular improve value given the configuration of the constraints on their variable(s). This allows an agent to estimate the likelihood that it has the best improve value among its neighbors in the absence of up-to-

date information. Using this likelihood as a basis for randomly determining when to change an agent's value, we end up with a DSA-like protocol where each agent's probability p_i is dictated by the improve distributions of its neighbors and its current improve value. This process is further enhanced by considering the use of explicit improve messages like those used in DBA. Unlike DBA, DPP sends out improve messages with a probability that is associated with its estimate of a neighbor having a prediction error of its improve value. This means that if agent X knows agent Y's improve PDF and agent Y behaves according to the protocol, agent X can even predict the probability that Y will have an improve value less than its own, even when Y has not sent X an improve message for a long period of time. As a result of these modifications, DPP uses considerably fewer messages than both DSA-B and DBA, does not require a user to define p values as in DSA, and more quickly converges to good solutions. The drawback of DPP is that calculating the initial PDF function can be problematic because it often does not have a closed-form solution.

The **Maximum Gain Message (MGM)** protocol [11] is a simplified version of the DBA protocol that does not change constraint costs to break out of local minima. MGM-2 is a variation that allows randomly selected agents to act as offerers and propose a coordinated move with one of its neighbors. Both of these algorithms monotonically improve the solution, which is often a desirable feature when the penalty for a failed coordination is high. One interesting feature of these algorithms is that we can calculate a bound on their optimality using k-optimality [24]. A k-optimal solution is one that cannot be improved without the coordinated action of k or more agents. The DSA, DBA, DPP, and MGM protocols are 1-optimal, MGM-2 is 2-optimal, and complete algorithms are n-optimal [10]. The KOPT

algorithm [10] takes this one step further by parameterizing the value for k. This implicitly allows a user to characterize their desired trade-off between computational expense and solution quality.

There are a number of other algorithms in this family, include the **Distributed, Penalty-driven Local search algorithm (DisPeL)** [2] and the **Distributed Simulated Annealing (DSAN)** protocol [1]. Essentially, each of these is a variant of the DSA or DBA protocol that use different methods for escaping local minima or managing asynchrony effects.

2.2.2 Complete Methods

The **Asynchronous Weak Commitment (AWC)** protocol is heavily based on its predecessor the **Asynchronous Backtracking (ABT)** protocol [30]. ABT works by assigning each agent a priority value. These priority values establish an absolute ordering amongst the agents that is used to control the search process. Agents perform the search by sending *value* messages to lower priority agents they are linked with. Value messages inform these lower priority agents about the variable values of higher priority agents. The agents use these values to determine if any of their domain values can satisfy their constraints with higher priority agents. Whenever the values of the higher priority prevent them from assigning their variable a conflict free value, the agent generates a *nogood* message.

Nogoods are composed of a set variable/value pairs that indicate that the combination of the variable assignments cannot be part of a satisfying solution. A nogood can be thought of as an implied constraint. After generating a nogood, it is sent to all the agent that are contained within it. Upon receiving a nogood, agents perform a linking step with any agent that is listed in the nogood and was previously unknown. This step is necessary to ensure

the completeness of the search. Initially, the linking structure mirrors the constraint graph, but because of linking as a result of nogoods, can quickly grow causing higher priority agents to send value messages to a large number of agents.

Like centralized backtracking algorithms, the ordering of the agents (variables) in ABT strongly affects the speed of the search. To overcome this problem, Yokoo created the AWC protocol [28]. The AWC algorithm is a variant of the ABT algorithm that allows the agents to re-prioritize themselves using the weak-commitment search heuristic [27]. This heuristic strategy basically says that whenever a backtrack occurs, that variable that triggered the backtrack should be moved up in the search tree. The principle idea behind this technique is to identify variables that are at the center of complex or critical constraints and assign them values first. Their values can then act as constraints on less critical variables instead of the other way around. In practice, this techniques has been shown to be quite effective in reducing the overall runtime of DCSP searches.

A later addition to the AWC protocol was the use of resolvent-based nogood learning [9]. This technique works by selecting, for each of the variable's possible values, one nogood that prohibits that value. These nogood are then merged together to form a new nogood. If the nogoods are selected wisely, they can actually generate more powerful nogoods that eliminate large portions of the search space.

Conceptually, **Asynchronous Partial Overlay (APO)** is based on the *cooperative mediation* paradigm [13]. Cooperative mediation entails three main principles. The first is that agents 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 prob-

lem. Third, agents should overlap the knowledge that they have to promote coherence. Together these three ideas create a powerful paradigm that has been applied to several distributed problems [17, 18].

The basic flow chart of the APO protocol is presented in Figure 1. 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 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. This step 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 that 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. Benisch and Sadeh, for instance, developed an inverted mediator

selection strategy, called Inverted APO (IAPO), that improves the parallelism of the protocol at the expense of requiring additional communication cycles [3].

When an agent links in APO, the agents exchange the domain values, D_i, and constraints, $\forall R_i\ x_i \in R_i$, on the their variable. In many environments, 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 directly exchanging this information is impossible due to privacy or security.

The **Asynchronous Forward Checking, Conflict-directed Backjumping (AFC-CBJ)** [20] protocol is a semi-synchronous adaptation of the Forward Checking, Conflict-directed Backjumping (FC-CBJ) algorithm [25]. It works by maintaining a token called the Current Partial Assignment (CPA). At any given time, the CPA is controlled by a single agent. When an agent has the CPA, it chooses a value from its current domain and adds it to the CPA. It then chooses one agent that is not in the CPA and forwards it to them. At the same time it sends forward check CPA (FC_CPA) messages to the other agents that are not in the CPA. If for some reason, the agent with the CPA does not have any remaining domain elements it sends a backtrack_CPA message to the last agent in the CPA with a conflict (a backjump). Upon receiving a FC-CPA message, agents remove any domain elements that are invalidated by the current assignments (a forward check). If they have no domain elements left, they send a Not_OK messages to every agent not in the CPA. Receiving a Not_OK message can trigger a backjump in the agent that holds the CPS. So essential, the assignment of variables is done in a synchronous manner, but forward checking is done asynchronously. Due to space limitations, we cannot fully describe the protocol in this paper.

Instead we refer the reader to the original article [20].

Like FC-CBJ, AFC-CBJ, has been shown to be a very efficient protocol in terms of the number of constraint checks it performs. However, it can use an exponential number of messages and communication cycles to solve a problem because of its synchronous, trial-and-error approach to evaluating assignments. Evaluation of the AFC-CBJ protocol is not included in this work because of the difficulty in comparing loss of privacy in structured, synchronous protocols to that of unstructured, asynchronous protocols.

3 Nogood-APO

The Nogood-APO (NAPO) algorithm is very similar in nature to the APO algorithm. The key difference is that instead of directly exchanging constraints, the agents exchange *nogoods* as part of the problem solving process. The procedures for NAPO are show in Figures 2 through 6.

By exchanging nogoods, the agents gain two things. First, because the agents incrementally reveal information, they may not have to reveal all of the details about 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 their constraints 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 it 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. The most important is that the agents need to maintain a *nogood list*. Like AWC, the size of the nogood list can grow quite large (exponential in the

worst case), especially if 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.

3.1 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 their direct nogoods and place them 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 NAPO, 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 consistent. This can be thought of as the agent deriving a unary nogood that 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 reasons 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 that is already in the list. This restriction ensures that the graph created by the variables in the good_list always remains connected.

3.2 Checking the agent view

Whenever the agent receives a message that 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. 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 messages 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.

3.3 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 two-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 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 with a higher priority agent or is expecting a request from an higher priority agent. 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 [6]. The goal of the search is to find a conflict-free solution for the variables in the session and to minimize the number of conflicts for variables outside the session (like the min-conflict heuristic [21]). During this search, new nogoods can be derived using nogood learning [7]. 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 its solution.

3.4 Example Execution

Consider the 3-coloring problem in Figure 7a. 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 red, green, or blue. 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 with whom they share constraints (their neighbors).

Once the initialization has completed, 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

  Green violates (ND0=G,ND1=G)

  Blue violates (ND0=B,ND3=B)

  Red causes no violations

- ND1

  Green cause no violations

  Blue violates (ND1=B,ND0=B) and (ND1=B,ND3=B)

  Red violates (ND1=R,ND2=R) and (ND1=R,ND4=R)

- ND2

  Green violates (ND2=G,ND1=G)

  Blue violates (ND2=B,ND3=B)

  Red violates (ND2=R,ND4=R)

ND3 computes a solution that changes the values of all of the variables in the session (see Figure 7b). Based on the information that ND3 obtained from the "evaluate!" messages, it believes that this solution solves its sub-problem and causes no conflicts for agents outside of the session. ND3 sends "accept!" message to the agents in the session.

After receiving the "accept" messages, each agent changes its value and checks its agent_view. This time, ND1 and ND2 are in conflict. This happened because ND3 changed their values to blue, inadvertently causing the violation. 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.

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

- ND0

  Green violates (ND0=G,ND3=G)

Blue violates (ND0=B,ND1=B)

Red causes no violations

- ND2

  Green violates (ND2=G,ND3=G)

  Blue violates (ND2=B,ND1=B)

  Red violates (ND2=R,ND4=R)

- ND3

  Green causes no violations

  Blue violates (ND3=B,ND1=B)and (ND3=B,ND2=B);

  Red violates (ND3=R,ND0=R)

- ND4

  Green causes no violations

  Blue violates (ND4=B,ND2=B)and (ND4=B,ND1=B)

  Red causes no violations

ND1 computes a solution which changes its value to green and ND2's to red and sends "accept!" messages. All of the agent's check their agent_view and find no conflicts, so the problem is solved (Figure 7c).

3.5 Soundness and Completeness

The soundness and completeness of the NAPO algorithm are derived directly from the soundness and completeness of APO. The reader is referred to [14] and [8] for the complete details of the proofs for APO. The original proof of APO's soundness and completeness was originally discovered to be incorrect in early 2005. In particular, during the development of an APO variant called Dynamic APO [15], an oscillation occurred during several of the test

runs. It was quickly determined that the oscillation occurred because of the inability of higher priority mediator to *always* achieve locks over lower priority mediators. The problem was quickly corrected by introducing a simple preemption mechanism that bumps lower priority agents. The reference implementation was corrected and being such a minor change the correction was never published.

The error was rediscovered in 2007 by Grinshpoun and Miesels [8] who had reimplemented the original version of APO from published descriptions instead of using the reference implementation. During their investigation, however, they were also able to identify another potential condition that may occur due to delayed state updating at the end of a mediation session. They corrected the problem by including all of the updated variable/value pairs as part of the accept message. Strangely, this information has always been included as part of the accept message in reference implementation, but was left out of the algorithm description because it was viewed as an implementation optimization.

Here is a basic outline of the proof for NAPO:

- 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 and live-lock free. The other half of the soundness proof.

- 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 from both "evaluate!" messages and "ok?" messages. This ensures completeness.

4 Empirical Evaluation

4.1 Experimental Setup

To test the NAPO algorithm, we implemented the AWC, APO, and NAPO algorithms and conducted experiments in the distributed 3-coloring domain. The particular AWC algorithm we implemented can be found in [32], which includes the resolvent *nogood* learning mechanism described in [9]. We conducted two 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 [5]. 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. Each data point represents an average over 30 randomly generated problems. Each algorithm was given the same problems with the same initial variable assignments to minimize variance. The algorithms were allowed to run for up to 1,000 cycles. This upper limit only affected the AWC protocol, which frequently could not finish on larger, higher density problems. A total of 2,250 test runs were conducted.

Phase transitions are important because that are strongly correlated with the overall difficulty of finding a solution to the graph [4, 22, 5]. Within the phase transition, randomly created instances are typically difficult to solve. Interestingly, problems to the right and left of the phase transitions tend to be much easier.

In 3-colorability, the value $d = 2.0$ is to the left of the phase transition. In this region, randomly created graphs are very likely to be satisfiable and are usually easy to find solve. At $d = 2.3$, which is in the middle of the

phase transition the graph has about a 50% chance of being satisfiable and is usually hard to solve. For $m = 2.7n$, right of the phase transition, graphs are more than likely to be unsatisfiable and, again, are also easier to solve.

During all of the tests we measured the number of messages, cycles, and non-concurrent constraint checks (NCCCs) [19] 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. We also instrumented the algorithms to measure the number of non-concurrent constraint checks used during each cycle. This measure has gained popularity in the DCSP community because it provides an implementation independent view of the parallel computation usage of a protocol by accounting for the amount of computation done during each of the cycles.

In addition to these standard measures of computation and communication cost, we also gathered data to quantify the information that the agents revealed to one another during the problem solving process. One measure we used was to count the number of links that the protocols created during execution. This metric provides insight into "who" the agents send information to in order to solve the problem. We also wanted to measure "what" and how much information was being sent. To do this we used the following measure of information gain:

$$gain(a_i) = \sum_{ng \in nogoods - rcvd_i} \frac{1}{|D_i|^{|ng|}} \tag{1}$$

where a_i is an agent, $nogoods - rcvd_i$ is the set of unique nogoods that have been received from other agents by a_i, $|D_i|$, is the size of the domain, and

$|ng|$ is the size of an individual nogood based on the number of variable/value pairs it contains. The logic behind this equation is that the power of a nogood can be measured based on the number of potential solutions that it invalidates in the search space. Shorter nogoods are more powerful because they are more general and eliminate a larger number of value combinations. This metric is similar to the Value of Possible States (VPS) metric developed by Meheswaran et al. [12]. In environments where the agents have different sized domains, this equation would need to be corrected by either using the maximum domain size of any variable or by unrolling the denominator in the equation to account for it.

One should keep in mind that this metric does not account for overlapping nogoods, but it does account for subsumed nogoods. In other words, if an agent receives two distinct nogoods that eliminate overlapping portions of the search space, that overlap is counted twice. However, if the solutions eliminated by a nogood are entirely eliminated by another nogood (subsumed by) then that nogood is not counted. For both of these metrics, we determined the average across the agents, measuring the distribution of information gain, as well as the maximum value for any single agent, measuring the amount of centralization.

To provide a frame of reference, we also included data for the average and maximum information gain for the case where the agents elect a leader and centralized the entire problem. The centralized maximum and average information gain can easily be computed as where n is the number of agent and m is the number of constraints:

$$max\_gain(a) = \frac{m(n-1)}{n * |D_i|} \tag{2}$$

$$avg\_gain(a) = \frac{max\_gain(a)}{n} \tag{3}$$

4.2 Results

The result of the phase transition experiments can be seen in Figure 8. These graphs show that AWC outperforms both APO and NAPO on very sparse problems, but on problems at or above the phase transition, the story is quite different. APO uses the least number of NCCCs, cycles, and messages with NAPO using slightly more. These results seem to contradict the findings of presented by Grinshpoun and Meisels [8] who reported that APO used more NCCCs on medium density problems across various levels of constraint tightness. The discrepancy between these results can likely be explained by the difference in the experiments that were conducted. Grinshpoun and Meisels used general CSP instances where the variables have large domains ($|D_i| = 10$) as opposed to the small domain of the variables ($|D_i| = 3$) and fixed tightness of the constraints ($p_2 = 0.33$) in 3-coloring. The large domains create equally large branching factors that severely impact the branch and bound solver used at the core of APO.

When looking at the results for information exchange, the nature of the protocols becomes apparent. APO, which uses explicit constraint passing, has the worse average information exchange across the entire transition, centralizes about 50% of the problem within a single agent, but creates the least number of new links. NAPO has the lowest average information gain, is equivalent to APO in the amount of information centralized in a single agent, and produces more links than APO. This can be interpreted as meaning that NAPO centralizes as much as APO, but does it in a more intelligent manner. AWC has the least average information gain on very sparse problems,

but within the phase transition performs worse than NAPO and actually approaches APO. AWC has very minimal centralization on sparse problems, but as the density increases, the agent with the maximum information gain actually gets more information than if the problem had just been completely centralized. At first, this doesn't appear to make sense, but AWC agents not only send original nogoods, they also send implied nogoods. So the agent with the maximum information gain is not only being told the other agents' nogoods, it is being told about nogoods that are learned by the other agents as well. AWC also creates more links meaning that agents are exchanging information with more of their peers than APO and NAPO

The results of the scalability experiments can be seen in Figures 9 and 10. The results for the cost metrics are as expected with APO using the least cycles, communication, and NCCCs of the three protocols. The protocol cost for NAPO is somewhere between APO and AWC. In the NCCCs category it appears that AWC and NAPO are competitive. However, one should keep in mind that many of the AWC runs did not actually complete on the 60 node test cases because they did not find a solution within 1,000 cycles. So the results for AWC in these graphs are skewed toward being lower than they actually are.

The results for scalability of information gain also present some interesting findings. They show that on sparse problem, both AWC and NAPO have less average information gain than APO. However, on denser problems, AWC becomes less scalable having a rapid increase in average information gain that exceeds even APO. The same trend holds true when looking at maximum information gain. AWC is dominate on sparse problems, but on dense examples has poor scalability. APO performs best overall in the number of new links it creates, with NAPO in the middle and AWC creating the

most links.

The take-home message from these experiments are not directly straightforward, but can be summarized as follows:

- On sparse 3-coloring problems, the AWC protocol exchanges the least amount of information in order to compute a solution, but takes more cycles, uses more messages, creates more links, and performs more NCCCs than APO.
- On dense 3-coloring problems, AWC exchanges more information, to more agents, uses more cycles, more messages, and more NCCCs than either NAPO or APO.
- If you are solely concerned about speed then APO is your best choice.
- If you are willing to trade speed for privacy than NAPO is the best choice on everything except very sparse problems.
- The speedups associated with partial centralization cannot be directly attributed to explicit constraint passing alone. Even when nogoods are exchanged, the algorithm performs as well or better than the distributed backtracking-based search.

4.3 APO and NAPO Variants

As mentioned previously, APO's parallel performance and degree of centralization are directly impacted by the heuristic that is used to select mediators. If the mediator is selected based on the agent that knows the most, then it should be expected that the protocol would experience a greater degree of centralization. The IAPO heuristic of choosing a mediator is based on who knows the least, however, so should show a wider distribution of knowledge

and a smaller degree of centralization. To test this claim, we repeated the experiments performed above to measure the performance of IAPO and NAPO with inverted mediator selection (INAPO).

The results of these experiments can be seen in Figure 11, 12, and 13 and are certainly interesting. As expected, the protocols that use the inverted selection heuristic use more cycles and messages across the entire phase transition and as the problem size increases. However, there are distinct bands present in the use of NCCCs based on whether the agents explicitly use constraints or not. The likely cause of this distinction lies in the nogood learning that is occurring in the central solver used by NAPO and INAPO.

Also interesting and rather unexpected, when looking at the average information gain, it appears that on low density problems that INAPO resembles NAPO, but near and beyond the phase transition, it resembles IAPO. One possible explanation for these results may be that on very sparse problems, it is best to use a technique that maximizes parallelism while avoiding information exchange. Keep in mind, however, that NAPO still has the least average information gain. When considering this in conjunction with the results for maximum information gain, it is clear that NAPO does very pointed centralization. Finally, this trend continues when looking at the linking that occurs during problem solving i.e. NAPO does more direct centralization, INAPO does a lot of information sharing, and IAPO, does minimal linking like APO.

The take-home message from these experiments can be summarized as follows:

- Using the inverted selection heuristic does not necessarily lead to more privacy.
- There are very few circumstances where the combination of inverting

the mediator selection and using nogoods provides a good solution.

- NAPO has the best privacy characteristics.
- APO is still computationally the fastest.

5 Conclusions

In this paper, we presented a new hybrid AWC/APO algorithm called Nogood-APO. As was shown in experimentation, this algorithm, like APO, outperforms AWC on all but the simplest 3-coloring problems across 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. In addition, we provided the first evaluation of the privacy afforded by using an inverse mediator selection heuristic and found that little additional privacy could be derived from this change.

Acknowledgement

The authors gratefully acknowledge support of the Defense Advanced Research Projects Agency under DARPA grants HR0011-07-C-0060. Views and conclusions contained in this document are those of the authors and do not necessarily represent the official opinion or policies, either expressed or implied of the US government or of DARPA.

References

[1] M. Arshad and M.-C. Silaghi. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, chapter Distributed Simulated Annealing. IOS Press, 2004.

[2] M. Basharu, I. Arana, and H. Ahriz. Solving discsps with penalty driven search. In *Proceeding of AAAI 05*, 2005.

[3] M. Benisch and N. Sadeh. Examining distributed constraint satisfaction problem (dcsp) coordination tradeoffs. In *International Conference on Automated Agents and Multi-Agent Systems (AAMAS)*, 2006.

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

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

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

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

[8] T. Grinshpoun and A. Meisels. Completeness and performance of the apo algorithm. *Journal of Artificial Intelligence Research*, 33:223–258, 2008.

[9] 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.

[10] H. Katagishi and J. P. Pearce. Kopt : Distributed dcop algorithm for arbitrary k-optima with monotonically increasing utility. In *Proceedings of the Ninth Distributed Constraint Reasoning workshop(DCR)*, 2007.

[11] R. T. Maheswaran, J. P. Pearce, and M. Tambe. Distributed algorithms for dcop: A graphical game-based approach. In *17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*, 2004.

[12] R. T. Maheswaran, J. P. Pearce, P. Varakantham, E. Bowring, and M. Tambe. Valuations of possible states (vps):a quantitative framework for analysis of privacy loss among collaborative personal assistant agents. In *Proceeding of Autonomous Agents and Multi-Agents Systems*, 2005.

[13] Mailler and Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 25:529–576, 2006.

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

[15] R. Mailler. Comparing two approaches to dynamic, distributed constraint satisfaction. In *Proceedings of Fourth International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2005)*, pages 1049–1056, 2005.

[16] R. Mailler. Using prior knowledge to improve distributed hill-climbing. In *Proceeding of IAT-06*, 2006.

[17] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceeding of AAMAS-2004*, pages 438–445, 2004.

[18] R. Mailler and V. Lesser. A cooperative mediation-based protocol for dynamic, distributed resource allocation. *IEEE Transaction on Systems, Man, and Cybernetics, Part C, Special Issue on Game-theoretic Analysis and Stochastic Simulation of Negotiation Agents*, 2006.

[19] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, 2002.

[20] A. Meisels and R. Zivan. Asynchronous forward-checking for discsps. *Constraints*, 12:131–150, 2007.

[21] 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.

[22] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.

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

[24] J. P. Pearce and M. Tambe. Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *International Joint Conference on Artificial Intelligence (IJCAI), 2007*, 2007.

[25] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):269–299, 1993.

[26] 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.

[27] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94); Vol. 1*, pages 313–318, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press, 1994.

[28] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Int'l Conf. on Principles and Practice of Constraint Programming*, pages 88–102, 1995.

[29] M. Yokoo and E. H. Durfee. Distributed constraint optimization as a formal model of partially adversarial cooperation. Technical Report CSE-TR-101-91, University of Michigan, Ann Arbor, MI 48109, 1991.

[30] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th Int'l Conf. on Distributed Computing Systems*, pages 614–621, 1992.

[31] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the 2nd Int'l Conf. on Multiagent Systems*, pages 401–408, 1996.

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

[33] W. Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*, 2002.

[34] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. Chapter 13: A comparative study of distributed constraint algorithms. *Distributed Sensor Networks: A Multiagent Perspective*, 2003.

[35] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 352–357, 2002.

6 Author Bio

Roger Mailler received his Ph.D. from the University of Massachusetts Amherst in 2004. After working as a postdoc at Cornell University, Dr. Mailler joined SRI International as a Senior Computer Scientist where he continued to research techniques for solving distributed problems. In 2008, he became an Assistant Professor at the University of Tulsa where he now directs the Computational Neuroscience and Adaptive Systems lab.

7 Figure captions

Figure 1: The flow chart of the APO protocol.

Figure 2: The APO procedures for initialization and linking.

Figure 3: The procedures for doing local resolution, updating the *agent_view* and the *good_list*.

Figure 4: The procedures for mediating a session.

Figure 5: Procedures for receiving a session.

Figure 6: The procedure for choosing a solution during an APO mediation.

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

Figure 8: Phase transition results for 30 node graphs of various density.

Figure 9: Scalability of cost results for AWC, APO, and NAPO.

Figure 10: Scalability of information results for AWC, APO, and NAPO.

Figure 11: Phase transition results using APO variants for 30 node graphs of various density.

Figure 12: Scalability of cost results using APO variants.

Figure 13: Scalability of information results using APO variants.

8 Figures

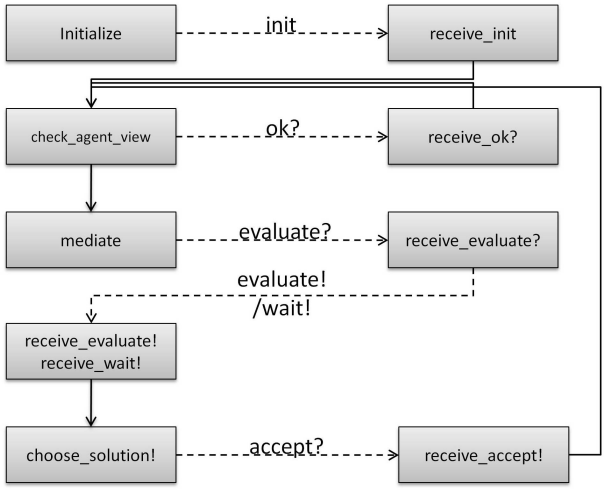

Figure 1: The flow chart of the APO protocol.

```
procedure initialize
   d_i ← random d ∈ D_i;
   p_i ← sizeof(neighbors) + 1;
   mediate ← null;
   add x_i to the good_list and agent_view;
   for each x_j ∈ neighbors do
      add x_j to good_list and agent_view;
      send_ok(x_j);
   end do
end

procedure send_ok(x_j)
   generate current confSet;
   if x_j ∉ agent_view do
      add x_j to initList;
      send (ok x_i, p_i, d_i, m_i, D_i, confSet) to x_j;
end
```

Figure 2: The NAPO procedures for initialization and sending ok? messages.

when received (**ok?** $x_j, p_j, d_j, m_j, D_j, confSet$) **do**
  if $x_j \in agent\_view$ **do**
    update $agent\_view$ with $(x_j, p_j, d_j, m_j, D_j)$;
  else
    add $(x_j, p_j, d_j, m_j, D_j)$ to $agent\_view$;
    if $x_j \in initList$ **do**
      remove x_j from $initList$;
    else
      send_ok(x_j);
  end if
  if $x_j \notin good\_list \wedge \exists_{ng \in agent\_view} x_j \in ng$ **do**
    add x_j to $good\_list$;
  add $\forall_{ng} \in confSet$ to $agent\_view$;
  check_agent_view;
end

procedure **check_agent_view**
  if $initList \neq \emptyset$ **do return**;
  $m'_i \leftarrow hasConflict(x_i)$;
  if m'_i and $\neg\exists_j(p_j > p_i \wedge m_j =$ **true**$)$
    if $\exists_{d'_i \in D_i}$ (d'_i does not conflict)
      and d_i conflicts exclusively with lower priority neighbors **do**
      $d_i \leftarrow d'_i$;
      for $\forall_{x_j \in agent\_view}$ **send_ok**(x_j);
    else
      do mediate;
  else if $m_i \neq m'_i$
    $m_i \leftarrow m'_i$;
    for $\forall_{x_j \in agent\_view}$ **send_ok**(x_j);
  end if
end

Figure 3: 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
   mediator ← x_i;
end

when received (wait!) do
   counter --;
   if counter = 0 do choose_solution;
end

when received (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) and nogoods ∈ labeled D_j;
   counter --;
   if counter = 0 do choose_solution;
end
```

Figure 4: The procedures for mediating a session.

when received (**evaluate?** x_j, p_j) **do**
  $m_j \leftarrow$ **true**;
  if *mediator* = **null** $\wedge \exists_k (p_k > p_j \wedge m_k =$ **true**) **do**
    send (**wait!**);
  else if *mediator* $\neq$ **null** $\wedge p_m > p_j$ **do**
    send (**wait!**);
  else
    mediator $\leftarrow x_j$;
    label each $d \in D_i$ with the set of nogoods
      that would be violated by setting $d_i \leftarrow d$;
    send (**evaluate!** $x_i, p_i,$ *labeled* D_i);
  end if
end

when received (**accept!** x_j, p_j, s) **do**
  update *agent_view* with (x_j, m_i, s);
  if *mediator* $= x_j$ **do**
    $d_i \leftarrow d'_i \in s$;
    mediator $\leftarrow$ **null**;
  end do
  for $\forall_{x_j \in agent\_view}$ **send_ok(**x_j**)**;
  check_agent_view;
end

Figure 5: Procedures for receiving a session.

```
procedure choose_solution
  select a solution s using a Branch and Bound 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;
  if mediate = x_i do
    d_i ← d'_i ∈ s;
  for each x_j ∈ agent_view do
    if x_j ∈ preferences do
      update agent_view for x_j
      if d'_j ∈ s violates an x_k and x_k ∉ agent_view do
        send_ok(x_k);
      end if;
      send (accept! x_i, m_i, s) to x_j;
    else
      send_ok(x_k);
    end if;
  end do;
  mediate ← null;
  check_agent_view;
end
```

Figure 6: The procedure for choosing a solution during an NAPO mediation.

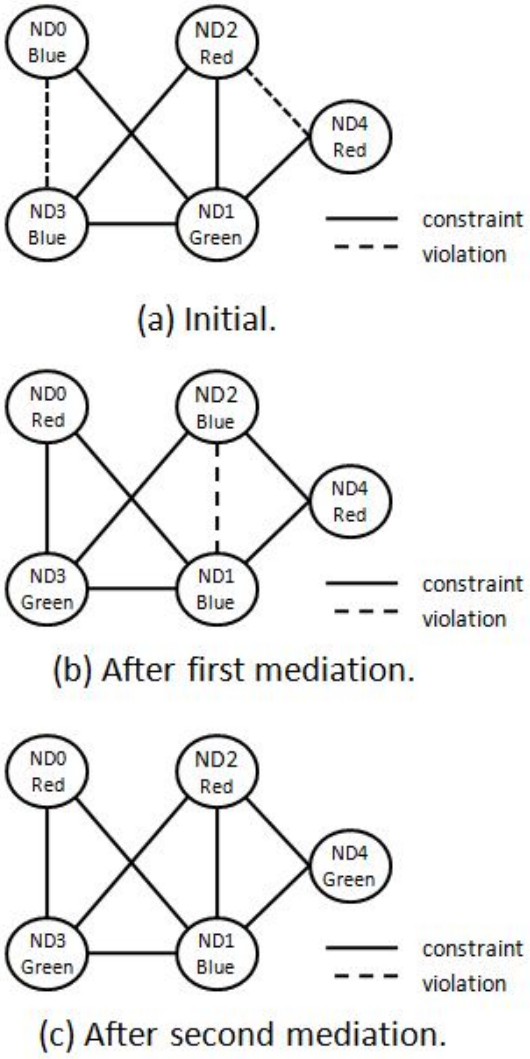

Figure 7: Example 3-coloring problem with 5 variables and 7 not-equals constraints.

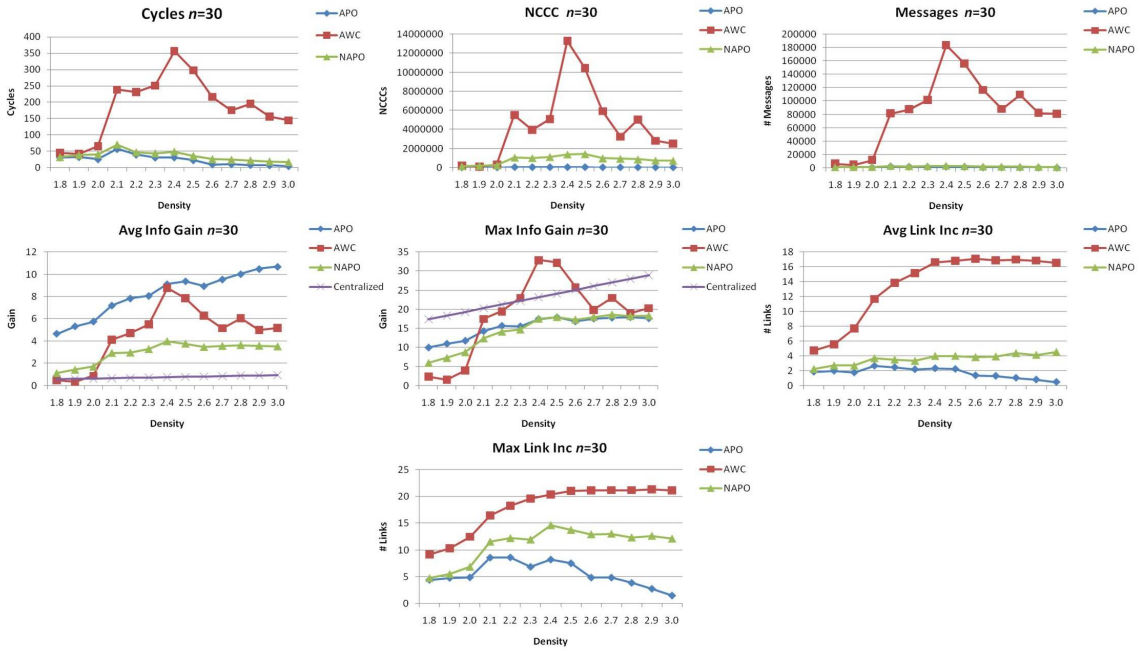

Figure 8: Phase transition results for 30 node graphs of various density.

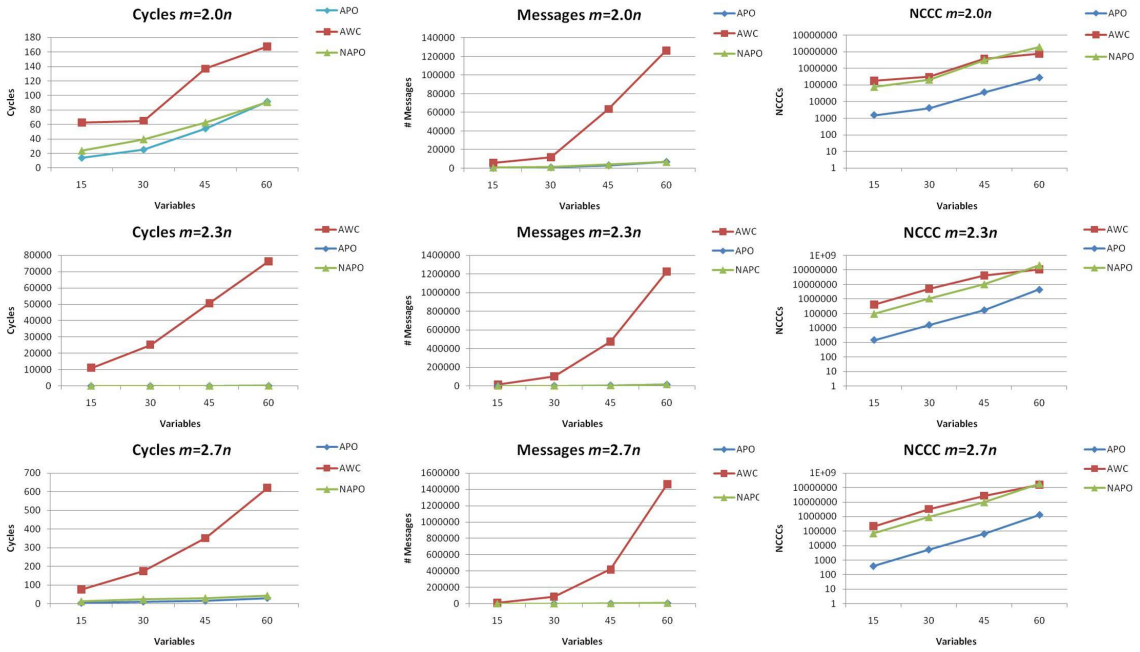

Figure 9: Scalability of cost results for AWC, APO, and NAPO.

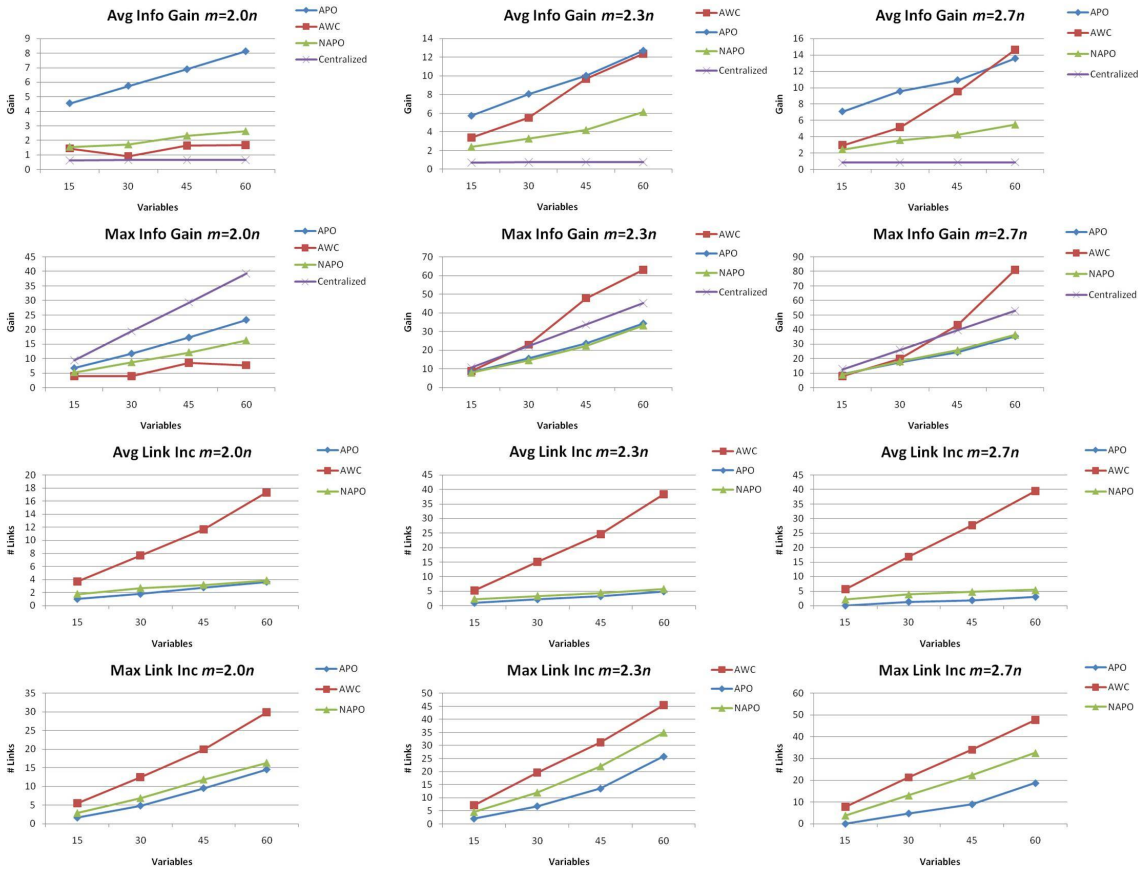

Figure 10: Scalability of information results for AWC, APO, and NAPO.

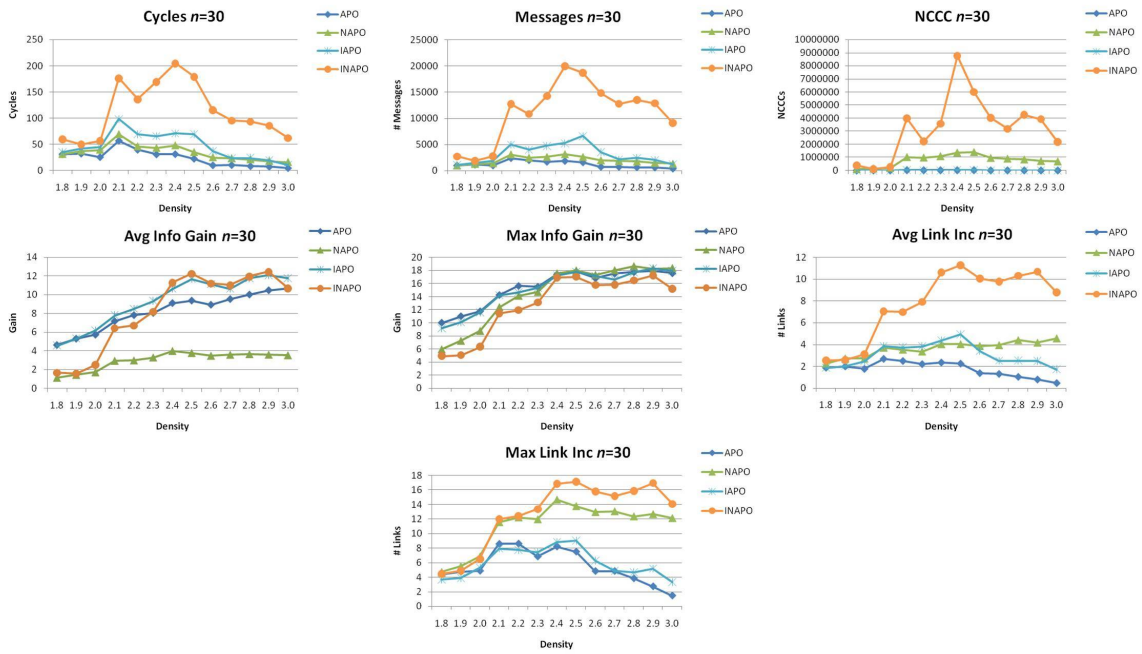

Figure 11: Phase transition results using APO variants for 30 node graphs of various density.

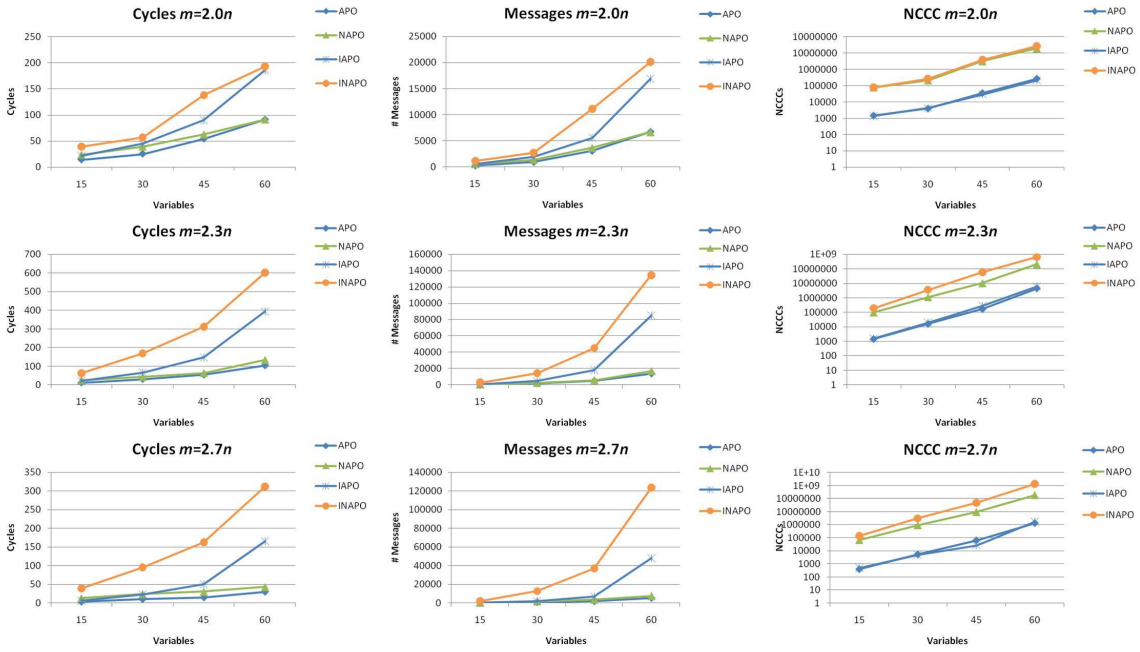

Figure 12: Scalability of cost results using APO variants.

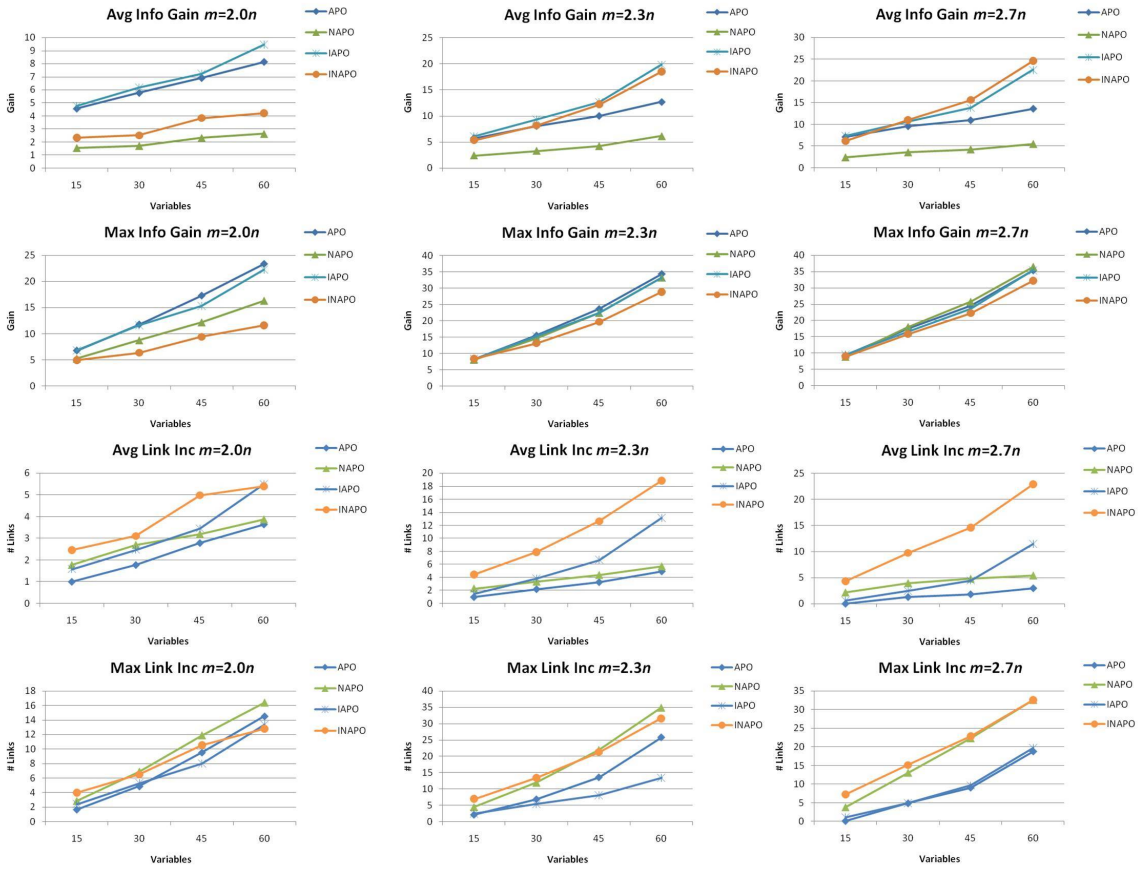

Figure 13: Scalability of information results using APO variants.