Algorithmic and Domain Centralization in Distributed Constraint Optimization Problems

John P. Davin

CMU-CS-05-154
July 2005

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Thesis Committee:

Pragnesh Jay Modi, Co-Chair

Manuela Veloso, Co-Chair

Stephen F. Smith

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Department of Interior-National Business Center (DOI-NBC).

Keywords: multiagent systems, distributed optimization, DCOP, Adopt, OptAPO

Abstract

A class of problems known as Distributed Constraint Optimization Problems (DCOP) has become a growing research interest in computer science because of its difficulty (NP-Complete) and many real-world applications (meeting scheduling, sensor networks, military planning). In this thesis we identify two types of centralization relevant to DCOPs: *algorithmic centralization*, in which a DCOP algorithm actively centralizes part (or all) of the problem structure, and *domain centralization*, in which inherent centralization already exists in the domain specification.

We explore algorithmic centralization by empirically studying Adopt and OptAPO, two DCOP algorithms which differ in the amount of centralization they use. Our results show that centralizing a problem's structure decreases communication overhead, but increases local computation. We compare the algorithms through our contribution of a new performance metric, Cycle-Based Runtime, which takes both communication costs and local computation time into account.

We then explore domain centralization by studying meeting scheduling, which has problem structure clustered at scheduling agents. We present a novel variant of Adopt, called AdoptMVA, which uses a centralized search within agents to take advantage of the partially centralized structure. We show that when agent ordering is controlled for, AdoptMVA outperforms Adopt in situations where communication costs are high. We contribute a Branch & Bound search heuristic which works well for meeting scheduling problems with multiple variables per agent. We also empirically experiment with meeting scheduling, showing that meeting size is in some cases a better indicator of solution difficulty than the number of agents in a problem.

Acknowledgments

I would like to thank my advisor Manuela for her always insightful perspective and guidance on the challenges present in calendar scheduling and its real-world applicability. Her passion for agent-based systems is an inspiration. I thank Roger Mailler for helpfully providing his implementation of OptAPO, which made possible much of my investigation of algorithmic centralization. I especially am grateful to Jay for his valuable advice, brainstorming conversations and ideas on problems encountered during my work. His knowledge of Adopt and DCOP and facility for teaching were instrumental to my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multi-agent systems are becoming a pervasive element of real-world computing applications. They have the potential to be much more robust and fail-safe than centralized systems, and apply naturally to a number of problems such as scheduling, military planning, and search and rescue. In many multi-agent systems, constraint optimization is one of the key capabilities required. For example, an office assistant agent would need to optimize meeting scheduling problems. Or, field-based agents for the postal service might need to optimize delivery schedules.

The Distributed Constraint Optimization Problem (DCOP) [1] provides a natural framework for handling these types of optimization problems which are by nature distributed across multiple agents. DCOP can model rich constraint interactions between agents, allowing it to be applied to many types of multi-agent optimization problems. Further, there exist several DCOP algorithms for optimally solving these constraint problems. Some algorithms partially or fully centralize the problem in order to use traditional search procedures on the problem. DCOP algorithms can be classified along a spectrum of *algorithmic centralization* ranging from fully distributed algorithms to fully centralized algorithms. A DCOP domain also can include inherent *domain centralization* in which agents naturally have control of multiple variables.

This thesis primarily seeks to address the questions:

- *How is performance influenced by the amount of algorithmic centralization used in a DCOP search?*

- *And, how can we take advantage of domain centralization which occurs in problems such as meeting scheduling?*

1.1 Distributed Constraint Optimization Problems

Constraint Satisfaction Problems (CSP) have been a longstanding part of computer science theory and applications. CSP is a model for problems in which a set of variables is constrained in some way by a set of constraints on the variables' values. A solution to a CSP is one which satisfies all the constraints.

CSP has traditionally been solved in a centralized fashion, on a single computer. However, this is fairly limiting because it does not extend well to large distributed networks. Multi-agent systems are becoming increasingly important given their many applications in business, military planning, team-based games and other areas. The Distributed Constraint Satisfaction Problem (DisCSP) [2] models these types of problems for a distributed framework. This allows us to solve CSP in a distributed system.

The constraint satisfaction model is still somewhat restrictive because it only applies to problems which can be represented in a way where success is a yes or no decision. Satisfaction is a boolean solution which often does not apply in real world situations - solutions can fall along a large range of qualities, where almost-perfect may be an acceptable answer.

In reality, we want to optimize the solution to a given constraint problem - we want to find the best possible solution, even if a fully satisfying solution is not possible. Constraint optimization, which is more general than constraint satisfaction, allows us to do exactly that. Distributed Constraint Optimization Problems (DCOP), as defined in [1], provide a formal model for optimizing a set of variables in a distributed manner.

A DCOP is defined as:

- set of N *agents*, $A = \{A_1, A_2, \ldots, A_N\}$.
- set of n *variables*, $V = \{x_1,x_2, \ldots, x_n\}$.
- set of *domains* $D = \{D_1, D_2,\ldots, D_n\}$, where the value of x_i is taken from D_i. Each D_i is assumed finite and discrete.
- set of *cost functions* $f = \{f_1, \ldots, f_k\}$ where each f_i is a function $f_i : D_{i,1} \times \cdots \times D_{i,j} \rightarrow N \cup \infty$. Cost functions are also called *constraints*.
- a *distribution mapping* $Q : V \rightarrow A$ assigning each variable to an agent. $Q(x_i) = A_i$ means that A_i is responsible for choosing a value for x_i. A_i is given knowledge of x_i, D_i and all f_i involving x_i.
- an *objective function* F, generally defined as the total cost of constraints for a given solution A: $F(A) = \sum_{x_i,x_j \in V} f_{ij}(d_i, d_j)$

An optimal solution to a DCOP is an assignment of values to the variables V such that total cost F is minimized. DCOP is known to be NP-Complete, making it a challenging and rich problem, particularly when we try to scale to large problems.

In some domains centralization of external constraints is undesirable because of privacy concerns. In meeting scheduling, a model called Private Events as Variables (PEAV) [3] takes into account the fact that human agents often do not want to share full calendar information with other participants.

1.2 Taxonomy of Centralization

> **Definition:** *Centralization* of a DCOP refers to the aggregation of problem information in a single agent. This aggregation results in a larger local search space at the agent. A problem can be fully centralized, or partially centralized if only certain parts of the problem are shared.

In this thesis we define two types of centralization:

- *algorithmic centralization* - a DCOP algorithm actively centralizes parts of the problem structure that are not already naturally centralized. This can allow the algorithm to use a centralized search procedure on the information that was centralized within an agent.

- *domain centralization* - the domain inherently has some of the problem structure centralized at each agent. In other words, the problem is presented to us already partially centralized. The meeting scheduling domain is an example in which structure is partially centralized, since agents can have control over multiple meetings. Each agent has knowledge of all the meetings within its own calendar and the constraints between those meetings.

These two types of centralization can be used to classify DCOP algorithms and domains along a spectrum ranging from fully distributed to fully centralized. Figure 1.1 graphically represents these two dimensions, with several algorithms and domains placed at locations along the spectrum. For example, Adopt is at the low end of algorithmic centralization because it doesn't actively centralize the problem. On the other hand, a DCOP algorithm that communicated the problem structure and then used a Branch & Bound search is a fully centralized approach. OptAPO is roughly in the

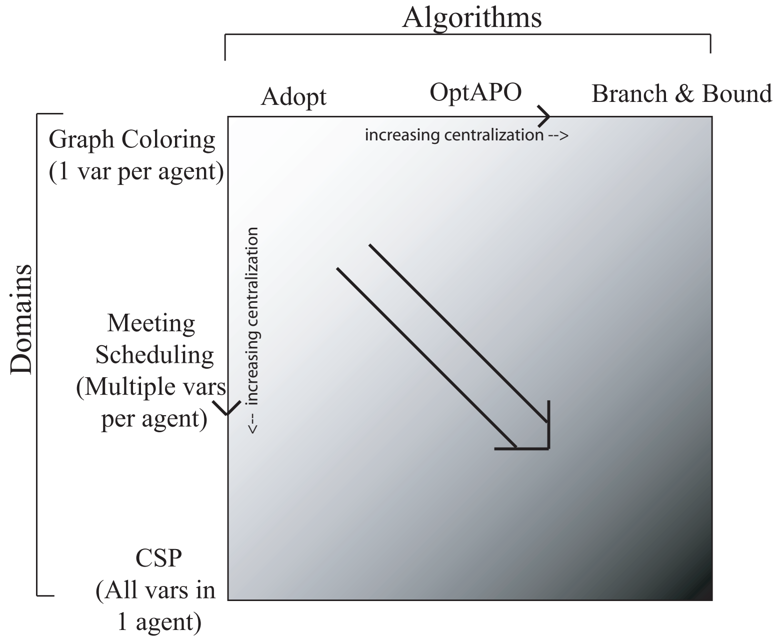


Figure 1.1: This figure shows the two dimensions of DCOP centralization - algorithmic and domain, along with labels marking approximately where certain algorithms or domains fall on the spectrum.

middle because it uses partial centralization. Along the domain centralization spectrum, graph coloring with one variable per agent is a fully distributed domain, while traditional CSP is a fully centralized domain. Meeting scheduling is somewhere in between because parts of the problem can be centralized.

Sometimes an algorithm handles domain centralization by reducing the problem into a fully distributed problem which can be solved as a typical DCOP. For example, the individual variables controlled by an agent can be represented as pseudo-agents [1] which each control a single variable and act independently.

1.3 Thesis Overview and Contributions

This thesis makes several contributions to our understanding of distributed constraint optimization problems:

- We define algorithmic and domain centralization, two dimensions along which DCOP algorithms and domains can be classified. This taxonomy provides a guide for classifying types of centralization, and emphasizes that centralization is an important aspect to consider in DCOP research.

- We explore both dimensions of centralization: Algorithmic centralization is studied by empirically comparing three algorithms which use different levels of centralization (Adopt, OptAPO, and Branch & Bound) [Chapter 2]. We study domain centralization by formulating a modified Adopt algorithm called AdoptMVA which uses partially centralized search to take advantage of problem structure [Chapter 3].

- We develop the CBR performance metric, which takes both communication cycles and local computation time into account. The metric provides researchers with a tool to more accurately compare performance of DCOP algorithms. [Chapter 2]

- We empirically study meeting scheduling problems, and find that meeting size can be an equally important performance factor as the number of agents in the problem. [Chapter 3]

The work here addresses the graph coloring and meeting scheduling domains. Some of the results may extend to other domains, and we will attempt to make note of ways in which they may or may not extend to other domains.

The data used in developing this thesis represent over 3000 runs of meeting scheduling problems, totaling over 800 hours of processor time, and close to the same in graph coloring. It therefore provides an unprecedented view of these domain's behaviors, particularly the effects of algorithmic and domain centralization on performance.

This document consists of two main sections. Chapter 2 illustrates the effect of algorithmic centralization by comparing three DCOP algorithms which differ in amount of centralization. We present results from graph coloring problems using a new metric which more accurately represents the runtime of distributed algorithms. Chapter 3 extends our interest in centralization by examining domain centralization which occurs naturally in problems such as meeting scheduling. A new algorithm based on Adopt is proposed for taking advantage of the problem structure that is naturally available to agents. A key difference between Chapters 2 and 3 is that the first deals solely with agents that control a single variable, while the second deals with agents that control multiple variables. Chapter 4 presents our conclusions and discusses possible future directions.

Chapter 2

Algorithmic Centralization in DCOPs

In this chapter we discuss several metrics which are used to evaluate DCOP algorithms, and formulate a new metric which addresses problems with the ones used previously. We then use this metric to compare three DCOP algorithms which differ in the amount of algorithmic centralization they utilize. We show the effects of centralization on algorithm performance, and provide an analysis of performance at several different levels of communication efficiency.

2.1 Evaluation Metrics

Often, DCOP algorithms are initially evaluated on graph coloring problems, since they provide a simple testbed for comparing performance. Graph coloring is a well studied domain and can be easily compared to prior results. We follow prior work by using 3-coloring problems and varying the number of variables and link density to affect solution difficulty.

Ideally, one would test DCOP algorithms in a truly distributed setup, which is the setting they are designed for in practice. However, there are several practical issues that make it difficult to test an algorithm fully distributed across a cluster of computers:

- hardware availability - researchers often do not have access to a cluster of a sufficient number of computers.
- communication variability - inter-agent communication latency may be more variable across a network than it would be within a single computer.
- inconsistency of execution - due to the large number of random execution paths that can be taken when executing an algorithm asynchronously, each execution will produce slightly

different performance times. This makes it difficult to give reliable performance estimates and comparisons to other algorithms.

The most realistic method for providing repeatable execution is to use synchronous timesteps. For example, the Multi Agent Survivability Simulator (MASS) [4] is an event-based simulator which can distribute agents across a cluster of computers, but the execution is synchronized in order to avoid consistency problems. They use a simulation pulse to represent time and to tell agents when to execute.

2.1.1 Cycles

The acknowledged method of measuring synchronous execution is with discrete *cycles* [5], where a cycle is defined as such:

> **Definition:** A *cycle* is defined as one unit of algorithm progress in which all agents, in parallel, process their incoming messages, perform any required computation, and send their outgoing messages. Importantly, a message sent in cycle i is not received until cycle $i + 1$.

Cycles are convenient for comparing DCOP algorithms because they are independent of machine speed, network conditions, and other factors external to the algorithm. They provide us with a method for measuring the amount of communication performed by an algorithm. However, this does not measure the amount of pure computation done by an algorithm. In other words, the total cycle count does not tell us anything about the length of a cycle or the total runtime of the algorithm.

On initial consideration it might seem that the amount of computation could be accurately measured by the process's runtime on a single computer. However, since the agents must take turns using the processor and cannot execute in parallel as they would in a distributed system, the runtime may not accurately reflect the actual distributed performance. If the agents solving the problem do not share the computational burden relatively evenly, then they will not take advantage of the parallelism of distributed problem solving.

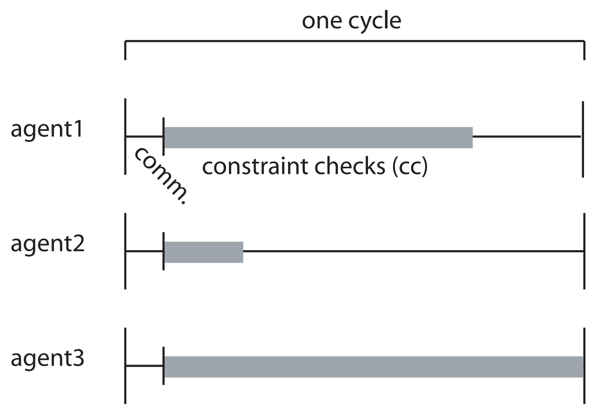


Figure 2.1: When agents execute in parallel, the length of the cycle as measured by concurrent constraint checks is determined by the maximum number of constraint checks of the agents.

2.1.2 Concurrent Constraint Checks

A common measure of computation in traditional constraint optimization algorithms is the *constraint check*, which is the act of evaluating a constraint between N variables. Constraint checks are considered representative of computational cost because they are the most basic operator of constraint optimization and scale in proportion to the size of the problem. When extended to distributed algorithms, this measure is called *concurrent constraint checks* [6] and is computed by selecting the maximum constraint checks from the agents during a cycle. The maximum is used because the length of a cycle in a distributed algorithm is determined by the slowest agent during that cycle (see Fig 2.1).

2.1.3 Cycle-Based Runtime

Given that synchronous cycles as discussed previously do not account for the local computation performed in a DCOP algorithm, we desire a metric that more accurately approximates the total runtime of an algorithm. Intuitively, we can capture an estimate of local computation costs by formulating a metric that includes concurrent constraint checks. We begin with a simple definition of runtime:

$$total\ runtime\ of\ m\ cycles = \sum_{k=0}^{m} time\ for\ cycle\ k \tag{2.1}$$

Now, we need a definition for the time of a cycle. A cycle involves communication followed by computation (see Fig 2.2). Let L denote the time required in a cycle to deliver all messages sent

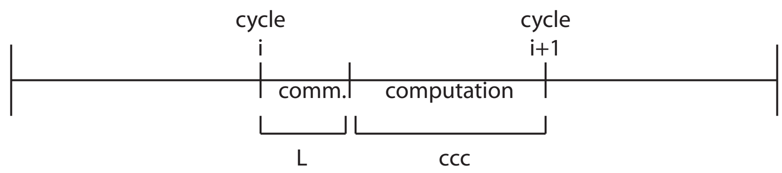


Figure 2.2: We model a cycle as being composed of communication (represented by L) and computation (measured with concurrent constraint checks).

in the previous cycle. We call this the *latency* of the underlying communication environment. L is algorithm independent. So we have:

$$time\ for\ cycle\ k = L + computation\ time\ in\ cycle\ k \tag{2.2}$$

We can now define the time to complete a cycle in terms of the number of constraint checks made in that cycle. Let $cc(x_i, k)$ be the number of constraint checks performed by agent x_i in cycle k. Then the computation time of cycle k is defined as:

$$computation\ time\ in\ cycle\ k = \max_{x_i \in V} cc(x_i, k) \times t \tag{2.3}$$

where t is the time required for one constraint check. t is a property of the underlying computing hardware and is algorithm independent. The max over all agents is used to compute concurrent constraint checks because the agents are conceptually executing in parallel. The length of a cycle is determined by how long the longest running agent took to complete. Substituting 2.3 into 2.2, we have

$$time\ for\ cycle\ k = L + \max_{x_i \in V} cc(x_i, k) \times t \tag{2.4}$$

Now substituting 2.4 in 2.1,

$$total\ runtime\ of\ m\ cycles = \sum_{k=0}^{m} (L + \max_{x_i \in V} cc(x_i, k) \times t) \tag{2.5}$$

Finally, the total number of concurrent constraint checks (ccc) performed by an algorithm over m cycles is defined as:

$$ccc(m) = \sum_{k=0}^{m} \max_{x_i \in V} cc(x_i, k) \tag{2.6}$$

Substituting 2.6 in 2.5, we arrive at our final equation for the time of m cycles, called Cycle-Based Runtime (CBR):

$$CBR(m) = L \times m + ccc(m) \times t \tag{2.7}$$

Note that the CBR metric is parameterized according to two environmental factors: the communication latency between cycles (L) and the speed of computation (t). Using this parameterized model, we can evaluate algorithm performance over a range of environments that vary in their relative speeds of communication and computation. Time required to transmit a message is usually greater than the time for a constraint check in most environments, so for simplicity we assume that a constraint check is the smallest atomic unit of time ($t = 1$), and assume L is given relative to t. We will explore four types of environments where communication costs are increasing by order of magnitude relative to computation, i.e., $L = t$, $L = 10t$, $L = 100t$, $L = 1000t$.

Note that CBR does not take into account number of messages or the time required to process messages. In other words, we assume that message processing time per cycle is not a significant differentiating feature between algorithms under comparison. We believe this is true for the algorithms compared in this paper. While Adopt uses many more messages than OptAPO, this is explained by its higher cycle count, i.e, the number of messages communicated per cycle is about the same between the two algorithms. Also, we assume the time to process each message is similar for both algorithms.

2.2 DCOP Algorithms

We used two DCOP algorithms in our work - Adopt, and OptAPO. We will provide background on the algorithms and explain how they differ.

2.2.1 Adopt Algorithm

Adopt [7], or Asynchronous Distributed OPTimization, is a complete and asynchronous DCOP algorithm developed by Jay Modi et al. The algorithm is fully general and can work with unary, binary, and n-ary constraints.

Adopt is a backtracking search that maintains a lower and upper bound at each variable during its search. It progressively narrows the range between them in order to arrive at the optimal solution, and terminates execution when the lower bound equals the upper bound at the root agent. Adopt constructs a priority ordering of the variables during an initialization phase. This ordering, which can be structured as a tree or as a simple chain, determines the parent and children of each variable.

Agents communicate their current value to all neighboring agents lower in the priority tree by passing down VALUE messages (agents are neighbors if they have a constraint between them). After locally computing a lower and upper bound based on available knowledge, an agent sends a COST message up to its parent. The COST message contains the newly computed lower and upper bounds, and the variable context that those costs are dependent on (variables that were used to compute the cost).

The stored costs can then be used in future iterations to develop a more accurate estimate of the optimal solution cost. These costs are dependent on the values of the agent's ancestor variables at the time the cost was computed. Therefore, when an ancestor changes its value, the costs dependent on it become invalid and are deleted. This can impact solution time because costs need to be recomputed; Adopt's variable ordering heuristic is intended to reduce the negative impact of these context switches. The variable ordering used by Adopt has a large impact on search difficulty, and thus it is important to choose a good order. The importance of variable ordering was observed experimentally in this thesis and will be discussed in Chapter 3.

2.2.2 OptAPO Algorithm

OptAPO [8], or Optimal Asynchronous Partial Overlay, is an alternative complete and asynchronous DCOP algorithm designed by Mailler and Lesser. The algorithm uses an approach termed *cooperative mediation* in which an agent is dynamically chosen as a mediator and temporarily put in charge of collecting constraints for a subset of the problem. OptAPO thus partially centralizes the problem within the mediator, and then uses a centralized search to optimize the subproblem.

Election of the mediator is done in an intelligent way using dynamic priorities determined during problem solving. The mediator uses the centralized Branch & Bound search of Freuder and Wallace [9] to compute the optimal solution for the variables and constraints it has knowledge of.

Agents in OptAPO use a novel cost justification technique to drive the communication of constraints. This technique avoids centralization when it is deemed unjustified based on problem structure. As an OptAPO agent receives constraints from other agents in the problem, it adds the other agents to a data structure called its *goodlist*. The goodlist is a list of all the agents centralized within a given agent during problem solving, and we will later use this to measure the amount of centralization in OptAPO.

2.2.3 Level of Centralization in Adopt and OptAPO

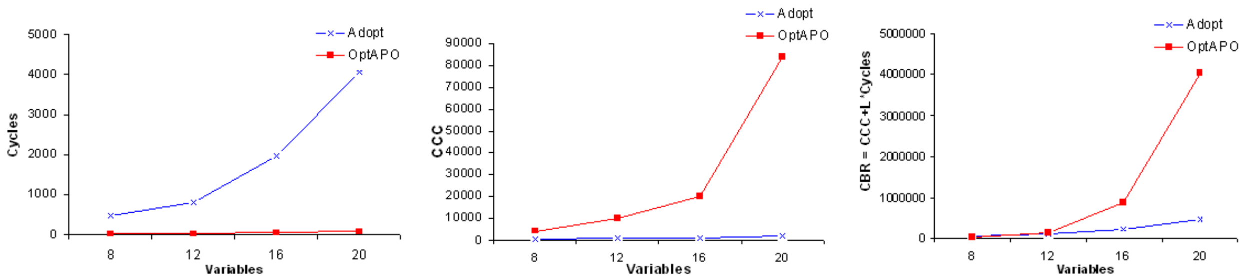
A key difference between Adopt and OptAPO is the level of algorithmic centralization each use. OptAPO communicates information about variables and constraints that are not directly connected to the agent - i.e., the agent gains information about variables that are not within its set of neighbors. This relies on the assumption that an agent can communicate with all other agents in the problem. While in practice this is not true of all domains, it can be accomplished by using a multi-hop message passing strategy, at the cost of increased communication.

OptAPO's centralization gives an agent broader knowledge of the problem, potentially allowing it to take advantage of this in its local optimization. Adopt on the other hand, does not centralize problem structure because agents only use knowledge of their direct neighbors, which can be assumed to be freely available. Adopt and OptAPO are two points on a spectrum of centralization, with Adopt at the no centralization end, a fully centralized algorithm like Branch & Bound at the other end, and OptAPO somewhere in the middle.

This centralization property has significant implications on load balancing and the amount of computation that each agent must perform during problem solving. In the case of OptAPO, as the size of an agent's subproblem grows, more local computation (search) is required to find the optimal solution to the larger subproblem. In OptAPO, we may expect that the computational load at some agents will grow as problem solving progresses and their sub-problems grow. On the other hand, in an algorithm which does not communicate constraints, such as Adopt, we may expect that the computational load at each agent will remain constant during problem solving.

2.3 Results

We evaluated Adopt and OptAPO in a simulator framework instrumented to measure concurrent constraint checks and cycles. Following previous work [1, 8], we then ran OptAPO and Adopt on a set of randomly generated 3-coloring problems. The problems were generated with problem sizes of n=8, 12, 16, or 20, and a link density of either 2n or 3n. Each problem size had 50 generated problems (for a total of 8*50 = 400). The same set of randomly generated graphs was used for each algorithm.



(a) Number of Cycles (b) Concurrent Constraint Checks (c) Cycle-Based Runtime

Figure 2.3: Main result: (a) OptAPO requires fewer cycles than Adopt, as shown in previous research, (b) But requires an increased amount of computation as measured by constraint checks. (c) When both constraint checks and communication latency (with L=100) are accounted for, Adopt outperforms OptAPO.

2.3.1 Cycle-Based Runtime of Adopt and OptAPO

Constraint checks and cycle counts were logged and used to compute the value of CBR in Equation 2.7 for four different values of L. As described in Section 2.1.3, L represents the time required by the communication environment to deliver messages between cycles specified relative to the time for a constraint check. For example if $L = 1$, we are assuming communication is very fast and on the same order of magnitude as a constraint check. If $L = 1000$, we are assuming communication takes three orders of magnitude longer than a constraint check.

Our experiments showed that OptAPO completes in fewer cycles than Adopt (see Fig 2.3a), as would be expected given prior research [8] and the fact that OptAPO is partially centralized. However, from Figure 2.3b we see that OptAPO actually requires many more constraint checks than Adopt and this results in a higher CBR at L=100.

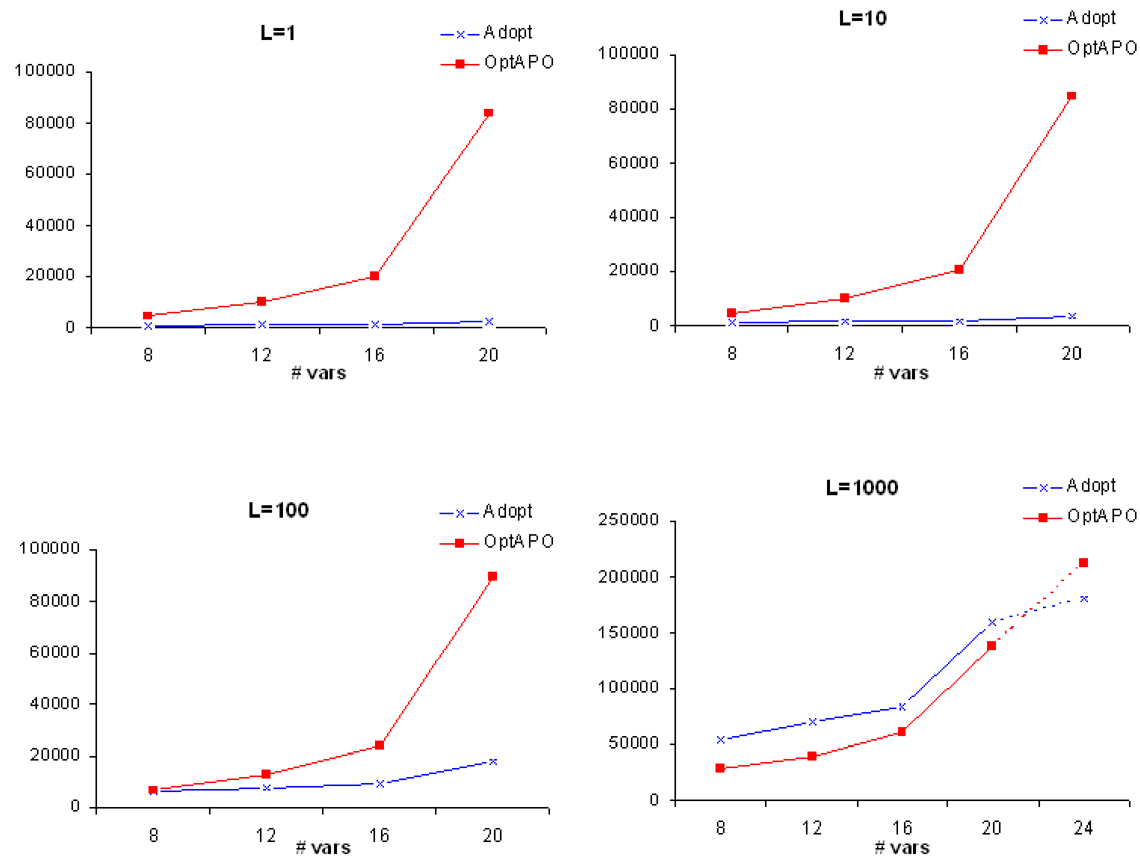


Figure 2.4: Comparison of Adopt and OptAPO using the CBR metric on graphs of low density. Each graph represents a different L value.

Figures 2.4 and 2.5 show four graphs generated from a single set of experiments on problems of link density 2n and 3n respectively. Each datapoint represents the average of the 50 problems. In Figure 2.4, we see that when L is 1, 10, and 100, Adopt outperforms OptAPO. At $L = 1000$, Adopt performs slower than OptAPO on the four problem sizes we tested. However, we extended the experiment to 24 variables for a smaller set of problems (20 problems at density 2, and 10 at density 3). We used a smaller dataset because the large problems have much longer runtimes. The performance on these problems has been shown with a dotted line on the $L = 1000$ graph, and indicates that Adopt may outperform OptAPO on large problems even at $L = 1000$.

We conclude that while Adopt requires more cycles than OptAPO, each OptAPO cycle takes significantly longer than each Adopt cycle. L provides a parameter to vary the relative cost between number of cycles and length of each cycle. For a significant range of L, Adopt performs better than

OptAPO, and as problem size grows this range increases.

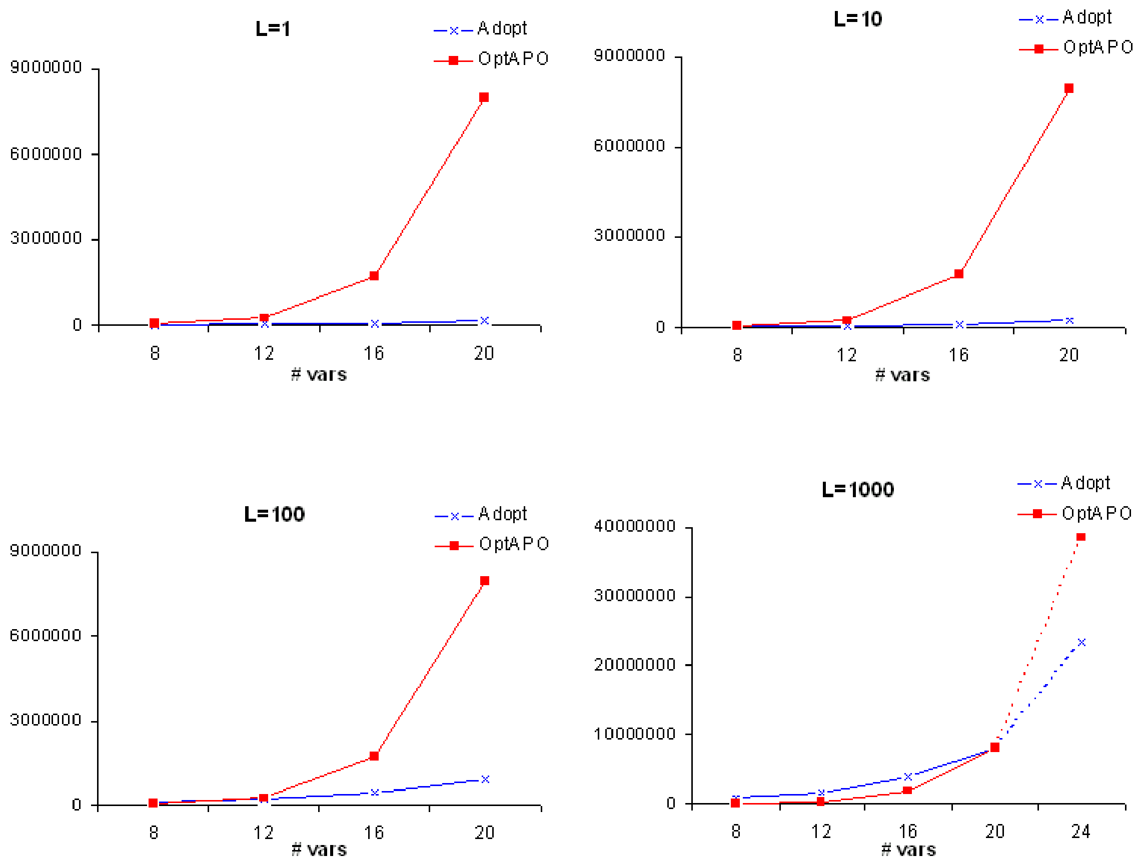


Figure 2.5: Comparison of Adopt and OptAPO using the CBR metric on graphs of high density. Each graph represents a different L value.

2.3.2 Centralization in OptAPO

We have hypothesized that the degree of centralization is the reason that OptAPO's cycles take much longer than an Adopt cycle. To verify this, we recorded the amount of centralization that the OptAPO agents reached by termination, as represented by the size of the OptAPO *goodlist*, which contains the other agents whose constraints have been centralized within an agent.

We computed the average, minimum, and maximum goodlist sizes across the agents in a problem at termination. We obtained similar results to the centralization data reported in Mailler's thesis [10]. As seen in Figure 2.6, on low density problems OptAPO agents on average have centralized at least half of the problem by the time a solution is found. On highly dense graphs, which are more difficult and time-consuming to solve, OptAPO on average centralizes nearly all of the problem.

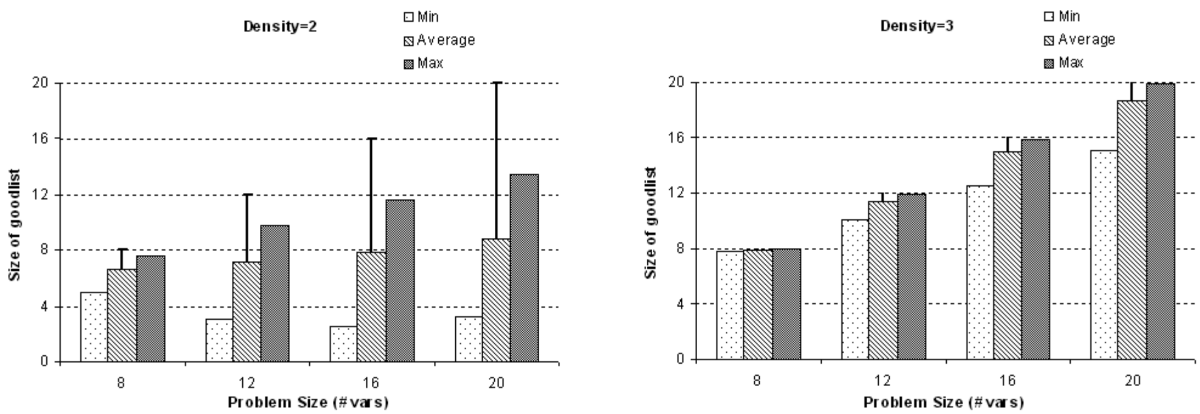


Figure 2.6: OptAPO centralization as measured by goodlist size. We display the minimum, maximum, and average amount of centralization of all the agents in a problem. The upper line above each bar marks n (# of variables), which is the maximum possible centralization at each problem size. Each measurement is the average of 50 problems.

The Max bars show that in high density graphs, almost all problems had at least one agent that fully centralized the problem. In low density problems, on average there was at least one agent who centralized about 75% of the problem. These results confirm our belief that OptAPO's centralization is a dominant feature of the algorithm which we believe explains the computational characteristics seen when computing CBR.

2.3.3 Distribution of Computation in Adopt and OptAPO

So far we have found that OptAPO does more computation, based on our measurement of the concurrent constraint checks performed across the agents during each cycle. However, we would also like to determine whether the higher maximum constraint checks is due to OptAPO simply doing more computation in *all* the agents during a cycle, or if it is due to uneven distribution of the computational load.

As discussed in Section 2.1.3, $cc(x_i, k)$ is the number of constraint checks performed by agent x_i in cycle k. Then, the distribution of computation within a cycle, which we will call $load(k)$, can be represented by the ratio of the maximum constraint checks to the total constraint checks in a cycle:

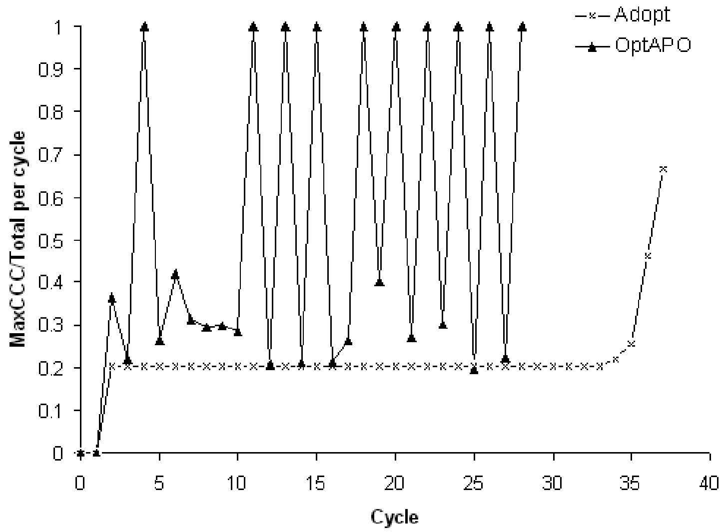


Figure 2.7: A measure of the distribution of computation in Adopt and OptAPO. The peaks on the OptAPO line indicate that in those cycles a single agent did most of the computation.

$$load(k) = \frac{\max_{x_i \in Agents} cc(x_i, k)}{\sum_{x_i \in Agents} cc(x_i, k)} \tag{2.8}$$

This equation represents the fraction of work that the maximum computing agent did during the cycle. A value of 1.0 means one agent did all of the computation in that cycle, and a lower value indicates the load was more balanced.

In Figure 2.7, the load ratio for OptAPO and Adopt is graphed for the execution of one representative graph coloring problem with 8 variables and a density of 2n. The x-axis is the execution time in cycles, and the y-axis is the load as defined in Eqn 2.8. The line for OptAPO shows spikes at cycles where an agent, the mediator, did a Branch & Bound search and accounted for most or all of the computation in that cycle. On the other hand, Adopt had very consistent distribution of computation, with most agents doing a similar number of constraint checks for most of the algorithm's duration.

This chart illustrates that OptAPO finished in a fewer number of cycles than Adopt, but the computation during those cycles is less evenly distributed among the agents, which results in longer time per cycle.

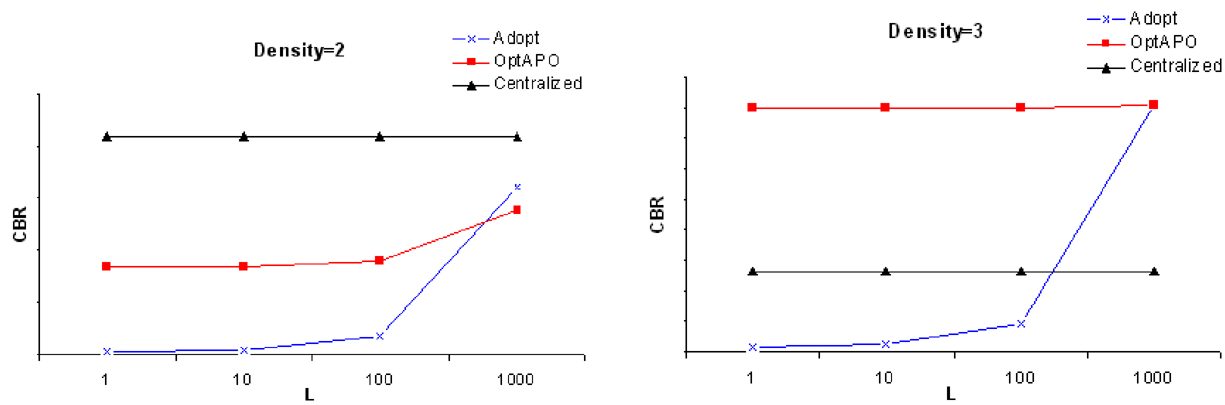


Figure 2.8: Adopt, OptAPO, and centralized search at 4 different L values. Each graph is based on 50 random problems of 20 variables.

2.3.4 Tradeoffs Between Communication Latency and Centralization

As our analysis has shown, a non-centralized algorithm like Adopt uses more communication cycles but has a lower computational cost per cycle. OptAPO, a partially centralized algorithm, has relatively low communication cycles but higher computational cost per cycle. We now ask how does a partially centralized approach like OptAPO and a decentralized approach like Adopt, compare with a completely centralized approach using CBR as an evaluation metric?

For the centralized approach, we assume one agent starts the algorithm with full knowledge of the problem, and simply invokes an optimization search procedure. We used OptAPO's implementation of centralized Branch & Bound search and measured the number of constraint checks required to find the optimal solution. We ignored the overhead cost that would be required in a truly distributed setting of electing a centralizer and all agents communicating the problem information to it. In the worst case, this cost is only some small factor of the width of the communication graph.

Figure 2.8 shows the three algorithms at different L values. As expected, the centralized algorithm is insensitive to varying L values because no communication is required. For both graph densities, Adopt is the best performing algorithm at L values less than 100. The crossover point occurs between L=100 and L=1000. These crossover points are important because they tell us at what point communication becomes too expensive for Adopt to operate efficiently, and tell us which algorithm should be used for a given communication environment.

For density 2, the OptAPO performance curve outperforms its own centralized solver using the CBR metric. These results agree qualitatively with the results using a serial runtime metric reported by Mailler and Lesser [8]. On density 3, the fully centralized approach had a lower CBR than OptAPO, which we believe may be explained by the fact that OptAPO does repeated multiple Branch & Bound searches, which could become more costly on dense graphs. The OptAPO searches partially reuse past searches, but this partial reuse does not completely recover the cost of the previous searches. From our analysis, we conclude that on high density graphs OptAPO eventually centralizes most of the problem, but does so with a higher cost than doing a simple centralization in the first step of the algorithm.

Figure 2.8 provides initial guidance to a researcher seeking to apply a DCOP algorithm to a new domain. The figure gives an estimate of which algorithm would be the most efficient for a given communication model and constraint density, although results in other domains may vary.

2.4 Conclusions

We have investigated two algorithms for DCOP - OptAPO and Adopt - that vary in the amount they centralize the problem in order to find the optimal solution. We developed a metric, CBR, for more accurately comparing these algorithms by taking into account communication latency between cycles and the length of each cycle. We have shown that while OptAPO requires fewer cycles than Adopt, OptAPO's cycles are longer because they require more computation. For domains with low communication latency compared to time to do a computation, Adopt outperforms OptAPO because in such domains agents are able to communicate efficiently and Adopt is able to take advantage of it by more evenly distributing the work of solving the DCOP. We have created graphs of the relative performance of Adopt, OptAPO, and centralized search under environments with varying communication latencies, providing the ability to choose the most effective level of centralization for each environment.

Chapter 3

Domain Centralization in DCOPs

As discussed in section 1.2, domain centralization is a feature of the problem as it is presented to us, and is essentially free to take advantage of since there is no loss of privacy and no communication is necessary. This led to our interest in determining whether a modified DCOP algorithm could benefit from the natural centralization in meeting scheduling.

In this chapter we present an alternative Adopt algorithm called AdoptMVA which takes advantage of domain centralization by using a local search procedure within agents that control multiple variables. The details of the algorithm are covered, and several heuristics for both the agent ordering and the intra-agent search ordering are compared. We show that when agent ordering is controlled for, AdoptMVA completes in fewer cycles than Adopt. We then present empirical analysis of the meeting scheduling domain showing its performance as the number of agents and meetings increases and as meeting size increases.

3.1 Motivating Domain: Meeting Scheduling

In many organizations, scheduling meetings among a group of people with busy schedules is a difficult and time consuming task. It usually involves a series of communications over email or phone to settle on a time that works for all required attendees. The negotiation becomes more complicated when participants have to bump around other meetings in order to accommodate the present one.

Furthermore, there are inherent privacy constraints in scheduling because some participants are unable or unwilling to reveal their entire schedule to others. In hierarchical organizations, members at certain levels may not want all of their available times to be known, and may have varying degrees of flexibility towards accommodating a meeting depending on how important it is. Given the natural distributed nature of this problem, distributed optimization algorithms are well suited for it.

Distributed optimization allows us to automatically schedule meetings using computer processes, and still preserve the privacy of a user's information. This would save valuable time in the workplace, and might even result in schedules that are more optimal than what humans would devise. However, there are a number of challenging issues that must be addressed first. Scheduling algorithms need to be fast enough to solve large optimization problems in a reasonable amount of time. This is a difficult problem given that scheduling is NP-Complete and grows exponentially as problems become larger. The good news is that there is a practical limit on the size of meeting problems that will need to be solved. It is unlikely that anyone will ever have an infinite number of meetings (though it might seem so), so we can assume an eventual limit on calendar size (eg., 100, or 1000). With the help of good DCOP algorithms and heuristics, optimal meeting scheduling may become a very tractable problem.

Another research area for scheduling is that algorithms need to be flexible enough to model the diverse types of scheduling situations that occur. Meetings often have varying levels of importance, attendance may be optional rather than required, and sometimes a substitute person (an assistant for example) can go in place of the requested person.

While the graph coloring domain has been explored fairly extensively with several DCOP solvers, meeting scheduling has not been explored as thoroughly. In some sense, the domain is more interesting because it has a slightly greater number of complexity dimensions which can be varied:

- N, number of agents
- M, number of meetings per agent
- D, domain size (number of hours available for scheduling)
- A, number of attendees per meeting (size of the meeting)

We have explored several of these dimensions, using Cycle-Based Runtime (CBR) [11] as the metric for comparing performance. Since the distributed algorithms were run in a synchronous mode on a single computer, CBR is appropriate for taking both the cycles and local computation into account.

The meeting scheduling domain is also interesting because it has a distribution of problem structure in which several variables (meetings) are clustered at individual agents. In other words, while graph coloring is generally treated in a fully distributed manner with each agent controlling only one variable, meeting scheduling agents control multiple variables. A primary focus of this chapter concerns exploring ways in which this domain centralization can be taken advantage of.

3.1.1 Multiagent Agreement Problem

Recent work in meeting scheduling has led to the development of a formal model for problems in which multiple agents must agree on a set of decisions. The Multiagent Agreement Problem (MAP) [12] is a special class of DisCSP [2] in which a variable can be shared among multiple agents. The model can be made equivalent to DisCSP however by simply giving copies of the variable to each agent and using inter-agent equality constraints to ensure agreement. The components of MAP are defined as follows:

- $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ is a set of *agents*.
- $\mathcal{V} = \{V_1, V_2, ..., V_m\}$ is a set of *variables*.
- $\mathcal{D} = \{d_1, d_2, ..., d_k\}$ is a set of *values*. Each value can be assigned to any variable.
- *participants*$(V_i) \subseteq \mathcal{A}$ is the set of agents assigned the variable V_i. The participants are responsible for choosing the value of V_i.
- *vars*$(A_i) \subseteq \mathcal{V}$ is the set of variables assigned to agent A_i.
- For each agent A_i, C_i is an *intra-agent* constraint that evaluates to true or false. It must be defined *only* over the variables in *vars*(A_i).
- For each variable V_i, an *inter-agent* "agreement" constraint is satisfied if and only if the same value from $\mathcal{D}$ is assigned to V_i by all the agents in *participants*(V_i).

We will use the MAP definitions to aid in explaining experiments described in this chapter. The MAP model can be used to represent the meeting scheduling problem, with the variables

$\mathcal{V}$ analogous to meetings, and the domain $\mathcal{D}$ equal to the set of timeslots that meetings can be scheduled for. The attendees of a meeting M_i are represented by $participants(V_i)$. *Inter-agent* equality constraints are used to ensure that attendees agree on the meeting start time, and inequality constraints within an agent (*intra-agent*) are used to ensure that none of the agent's meetings conflict.

3.2 Adopt with Multiple Variables per Agent (AdoptMVA)

As discussed in section 1.2, the current Adopt algorithm treats variables within an agent as pseudo-agents, running them in separate processes which are viewed as independent agents. However, this approach does not allow sharing of information beyond what Adopt already communicates and does not take full advantage of domain centralization.

A natural modification to this approach would be to create a single Adopt process for each scheduling agent, allowing it to control all of the meetings owned by that agent. Then, a local search procedure could be used to find the optimal local solution for the agent's calendar. We call this **AdoptMVA**, for Multiple Variables per Agent, since a single Adopt agent controls multiple variables. Figure 3.1 shows the new approach, in contrast to the standard Adopt pseudo-agent approach. The intuitive conjecture is that by breaking the problem up into smaller subproblems which are solved locally, the overall task could perhaps be solved quicker.

The algorithm was designed with meeting scheduling in mind, but could apply to other domains which have multiple variables per agent. We model AdoptMVA as an extension of the Adopt algorithm, but it requires a few new definitions in order to convert from a one variable per process paradigm to multiple variables.

In the Adopt formalism, a value assignment to a set of variables is called a context and is defined by Jay Modi in his thesis [7] as:

- Definition: A *context* is a partial solution of the form $\{(x_j, d_j), (x_k, d_k)...\}$. A variable can not appear in a context more than once. Two contexts are *compatible* if they do not disagree on any variable assignment. *CurrentContext* is a context which holds an agent's current knowledge of variable values for higher priority neighbors.

As defined in MAP, we let *vars*$(A_i) \subseteq \mathcal{V}$ denote the variables owned by agent A_i. We then

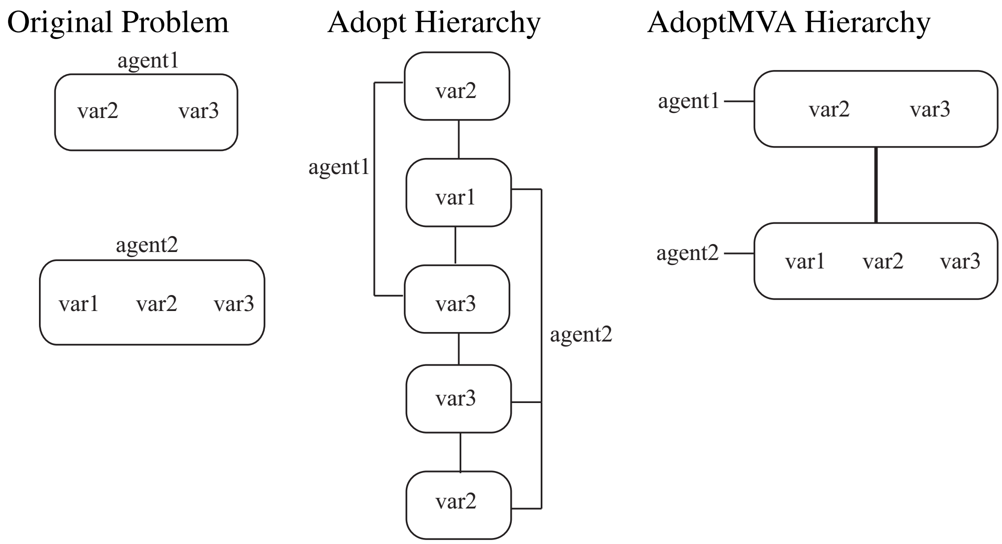


Figure 3.1: We show the original formulation of a meeting scheduling problem, and then a possible hierarchy of pseudo-agents that Adopt would use. We then show an agent hierarchy that could be used by AdoptMVA.

denote a context $s \in S$ where S is the set of all possible assignments to variables in $vars(A_i)$ and s is a particular one of those assignments.

Whereas previously the local cost of an agent was determined simply by its constraints with other agents, we now must define a function that also includes the cost of constraints between variables *within* an agent. We define the *local cost* δ for a particular value assignment s made by agent A_i for the variables it owns:

$$\delta(s) = \sum_{(x_j,d_j)\in s} \sum_{(x_i,d_i)\in s} f_{ij}(d_i, d_j) + \sum_{(x_j,d_j)\in CurrentContext} \sum_{(x_i,d_i)\in s} f_{ij}(d_i, d_j) \tag{3.1}$$

The first half of the definition sums the constraints between every possible pairing of variables within an agent, and the second half evaluates the constraints between each of the agent's variables and all of the external variables in its *CurrentContext*.

An agent computes a *lower bound for a solution s* using lower bound costs received from its children, denoted as $lb(s, x_l)$.

- Definition: $LB(s) = \delta(s) + \sum_{x_l \in Children} lb(s, x_l)$ is a *lower bound* for the subtree rooted at A_i when A_i chooses solution $s \in S$.

Similarly for an *upper bound on solution s*:

- Definition: $UB(s) = \delta(s) + \sum_{x_l \in Children} ub(s, x_l)$ is an *upper bound* for the subtree rooted at A_i when A_i chooses solution $s \in S$.

The overall lower and upper bounds for A_i are the minimums over the bounds for all possible solutions in S:

- Definition: $LB = min_{s \in S} LB(s)$ is a *lower bound* for the subtree rooted at A_i.

- Definition: $UB = min_{s \in S} UB(s)$ is an *upper bound* for the subtree rooted at A_i.

The above definitions are based on the original Adopt definitions [7] with the main change being that lower and upper bounds are now conditioned on a solution context rather than a single variable value. Also, the most important change to the local cost δ is that it now includes the cost of intra-agent constraints.

3.2.1 Details of AdoptMVA Algorithm

We implemented AdoptMVA by using the existing Adopt code and modifying it to assign all of an agent's variables to a single Adopt process. In order to properly communicate the values of an agent's variables to the other agents in the problem, we had to slightly modify Adopt's VALUE messages and its handling of COST messages.

VALUE messages must now include the values of all variables owned by the agent, rather than just a single variable value. Therefore, we extend VALUE messages to include a solution context:

- **Definition:** *VALUE(* $s_i = \{(x_j, d_j), (x_k, d_k), ...\}$ *)* is the form of the new VALUE messages sent to neighbors lower in the tree.

No change needs to be made to the content of the COST messages, but we do need to change the way receiving agents process them. The lower and upper bound costs reported in a COST message are now dependent on the aggregate of variables owned by the parent agent. Therefore we store children's costs attached to a solution context $s \in S$, which is equal to the set of the current agent's variables assigned to the values that were used when its child computed its reported cost. This previous assignment of values can be retrieved from the context reported in the COST message.

Note that the set of all possible solution contexts S is the set of all permutations of the values of the variables in $vars(A_i)$. The total number of permutations is equal to:

Algorithm 1: AdoptMVA Branch & Bound search

```
depth ⟵ 0
cost ⟵ 0
bestCost ⟵ ∞

procedure search( s ∈ S, depth, cost ):
if depth == s.size() then
    bestSolution ⟵ s
    bestCost ⟵ cost
    return;
end
V_i ⟵ variable at depth (order determined by variable ordering)
Domain ⟵ D_i
Reorder Domain with best-first heuristic (move current value of V_i in best known solution
to be first)
for d_i ∈ Domain do
    s{V_i} ⟵ d_i
    cost+ = Σ_{(x_j,d_j)∈s} f_ij(d_i, d_j) + Σ_{(x_j,d_j)∈CurrentContext} f_ij(d_i, d_j)
    if depth == s.size() − 1 then
        cost ⟵ cost + Σ_{x_l∈Children} lb(s, x_l)
    end
    if cost < bestCost then
        search( s, depth+1, cost )
    end
    s{V_i} ⟵ null
end
```

$$total\ solution\ contexts\ for\ agent\ A_i = |\mathcal{D}|^{|vars(A_i)|} \tag{3.2}$$

This number is potentially much higher than the number of solution contexts in traditional Adopt, which is limited to $|\mathcal{D}|$. From our experiments it appears that in some cases, this large number of contexts slows down the algorithm's progress.

3.2.2 Discussion of Branch & Bound search

The agent uses a local Branch & Bound search [9] to find the optimal solution to its variables, given the values of its ancestors in the Adopt hierarchy, and knowledge of the costs of its children. The search is used to calculate both the lower and upper bounds (LB and UB) on the agent's variables. Algorithm 1 shows pseudo code for the lower bound search (upper bound search is identical except $lb(s, x_l)$ is replaced with $ub(s, x_l)$).

Note that the cost function used within the search is not exactly the same as the δ function in Equation 3.1. Rather, cost for each variable is accumulated as the search proceeds down the tree, so that we don't redundantly calculate the entire cost at each recursion. The actual cost of δ is reached at the bottom of the recursion, and we then add in lower or upper bounds from our children to get the actual bound (if a bound for a child at the given solution s is not known, the lower bound defaults to 0 and the upper bound to ∞). Efficiency of the search could possibly be improved by attempting to apply the children's bounds higher in the search, to allow greater pruning of the tree. However, in order to compute a correct bound using a partial solution, the agent must have knowledge of all (or most) bounds for the set of possible solution contexts, which appears to happen fairly infrequently based on our observation of the algorithm.

We observed that although the algorithm must do a separate search for lower and upper bounds, there are some cases where these bounds are equal. We included some basic optimizations to avoid doing the upper bound search in certain cases. There are two cases where we skip the upper bound search:

- If the agent has no children in the hierarchy (it is a leaf), it's upper bound can automatically be set to equal the lower bound.
- If all known upper bounds received from children are already set to ∞, then the upper bound can automatically be set to ∞.

However, it would be possible to further improve the search. For example, a memoization structure could be used to cache results from previous cycles during the algorithm's progression. There are also known search heuristics which might further improve the Branch & Bound efficiency [13].

Despite not having the fastest possible search procedure, it was sufficient for producing a number of interesting experimental results, which we believe could be improved even more with faster search techniques.

3.2.3 Intra-agent variable ordering heuristics

We included two heuristics in the Branch & Bound algorithm to control: 1) ordering of variables, and 2) ordering of domain values. The value ordering is a simple best-first heuristic that puts the best domain value first in the search order for a given variable, where the best value is taken from the current best solution (if known).

The variable ordering heuristic determines the search depth at which each variable is optimized, and thus influences how much pruning can be done. We tested several heuristics, and present experimental results on them in section 3.3.3. The heuristics are:

1. **Lexicographic** - variables are ordering alphabetically.

2. **Random** - ordering is randomly selected each time a search is performed. Unlike the other heuristics we used, this order is different each time a search is executed and we therefore call it a randomly varying heuristic, as opposed to a static order.

3. **Brelaz** - This is based on the Brelaz heuristic [14]. We order by number of links to other variables within the agent, with higher link counts meaning higher ordering. We first compare number of links with already chosen variables, and if there is a tie we order by links with unchosen variables.

   This is only used for graph coloring. It is not useful for meeting scheduling with multiple variables per agent because within an agent, all agents have the same number of intra-agent constraints and the ordering would be arbitrary.

4. **MVA-AllVars** - order by number of links to external variables, considering *all* external variables in the problem. Higher link counts are ordered first.

5. **MVA-LowerVars** - order by number of links to external variables, considering only *lower priority* variables.

6. **MVA-HigherVars** - order by number of links to external variables, considering only *higher priority* variables.

These heuristics cover several qualitative dimensions as outlined in Table 3.1. The random heuristic is interesting because it is the only one that has a different ordering each time search is executed, and might have some success in finding solution paths that the other ones were not fortunate enough to find. We did not develop a heuristic that is both informed and randomly varies, but this would be an avenue for future work.

	informed	uninformed
static	MVA-*, Brelaz	Lexicographic
stochastically varying	-	Random

Table 3.1: A comparison of the differing characteristics of the intra-agent search ordering heuristics.

The MVA heuristics are novel as far as we know, and were designed to address the fact that the Brelaz heuristic is not applicable to meeting scheduling for AdoptMVA. In meeting scheduling, the MVA-AllVars heuristic mirrors meeting size, since the number of links a variable has to external variables is equal to the number of attendees in that meeting. This results in ordering large meetings at the top of the order, with smaller meetings at the bottom.

The other two MVA heuristics serve to investigate whether all external variables have impact on search cost, or if it is only lower priority or higher priority variables (where priority is determined by the AdoptMVA agent ordering). Our hypothesis was that the MVA-HigherVars heuristic would perform the best. In the AdoptMVA algorithm, higher priority variables are more influential on an agent's local search because the agent knows the values of higher priority variables (from VALUE messages) but does not directly know the values of lower priority variables (COST messages are only used at the bottom of the search). Section 3.3.3 presents experimental results which show that this hypothesis held true.

3.2.4 Inter-agent ordering heuristics

Another important performance factor for AdoptMVA is the agent ordering - i.e., the macro-level variable ordering which determines the Adopt hierarchy. We only tested chain orders; this simplified our analysis, and tree hierarchies have been studied in other research [3].

We present experimental results in section 3.3.2 for the following four heuristics:

- **Lexicographic** - Agents are ordered alphanumerically by agent name. This is an uninformed heuristic.

- **Inter-agent links heuristic** - this is a modification of Adopt's regular ordering, which orders by the number of links to chosen variables, and then by the number of links to unchosen variables. This heuristic is the same, expect we only count links to other agents (i.e., inter-agent links). It is analogous to the popularity of the agent - a person involved in many large meetings will be ordered higher because he will have more inter-agent links.

- **AdoptToMVA-Max** - Adopt's standard variable ordering is computed, and then converted into an agent ordering for AdoptMVA. The agent priorities correspond to the *maximum* priority variable within each agent in the Adopt ordering.

- **AdoptToMVA-Min** - This is the same as AdoptToMVA-Max except that agent priorities correspond to the *minimum* priority variable within each agent.

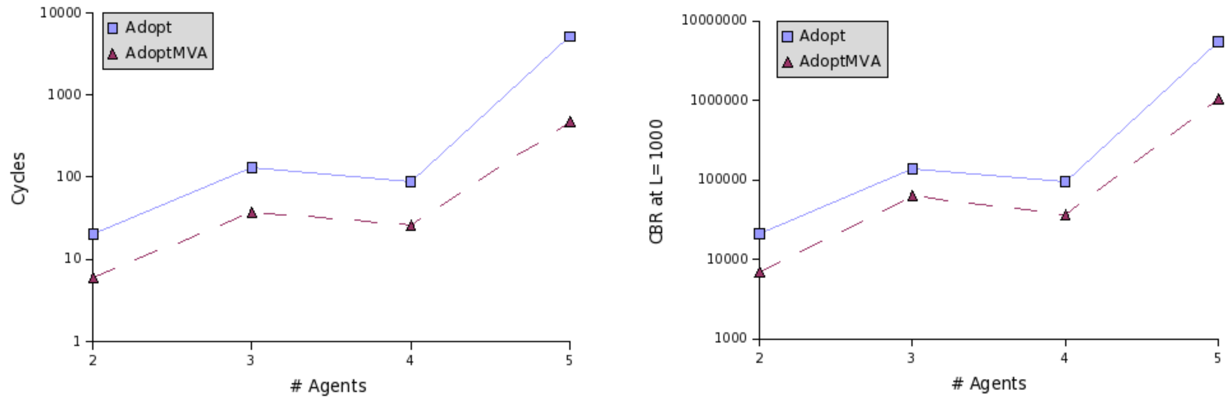
The AdoptToMVA orderings were created based on the theory that since the Adopt variable ordering works fairly well, converting it to an agent ordering might produce similar results for AdoptMVA. We convert an Adopt variable ordering to an agent ordering by assigning agent priorities based on the Adopt variable priorities (higher priority indicates higher placement in the chain). For an agent A_i, using AdoptToMVA-Max, $priority(A_i) = max_{\forall V_j \in vars(A_i)}\ priority(V_j)$. For the AdoptToMVA-Min heuristic we simply replace the max function in the above equation with min. The AdoptToMVA-Min ordering was introduced because initial experiments indicated that it sometimes performed better than AdoptToMVA-Max.

3.3 Results

We conducted an extensive number of experiments on both the graph coloring and meeting scheduling domains, with the goals of: a) determining the best heuristics for AdoptMVA, and b) gaining a more general understanding of meeting scheduling performance for a variety of DCOP algorithms.

We now discuss the experimental methodology used. Data in our results are based on the average of at least 20 randomly varied problems. Statistical t-tests are used to determine whether differences are significant, and we will present the results of these tests. We used scripts to generate problems which have a randomized constraint structure while keeping the problem size and density that was specified. For meeting scheduling problems, an 8 hour day is modeled by using 8 discrete timeslots, and the number of attendees in each meeting is randomly generated from a geometric progression. Unless otherwise noted, all meeting scheduling problems are fully schedulable (zero unscheduled meetings in the optimal solution) and graph coloring problems range from a solution cost of 0 up to 3 constraint violations.

The experiments were run on several Pentium 4 dual 3.0GHZ machines. We present results in terms of the Cycle-Based Runtime metric, which is independent of processor speed. As described in Equation 2.7, the formula for CBR is $CBR = Cycles * L + CCC$, where cycles is the number



(a) Cycles (b) CBR

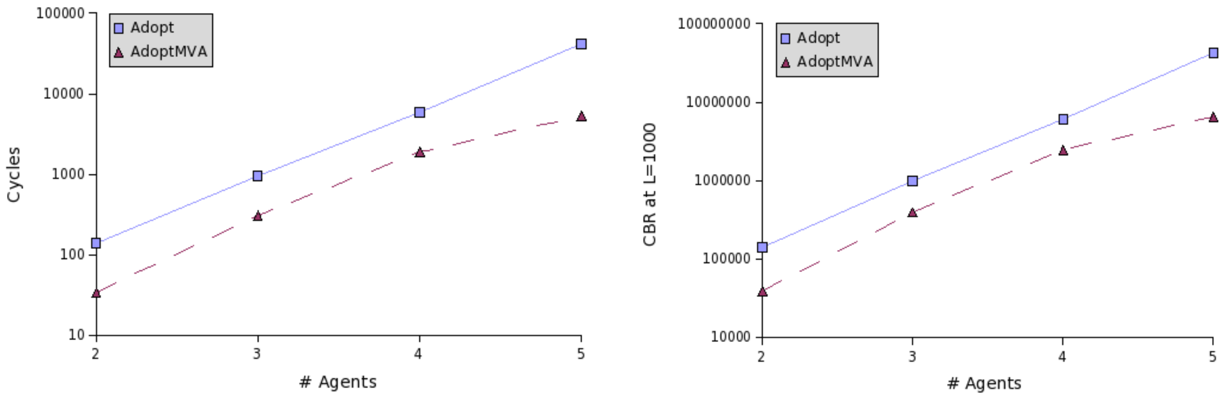
Figure 3.2: Using equivalent lexicographic agent orderings, on high density meeting scheduling problems with 4 meetings per agent, AdoptMVA outperforms Adopt in terms of cycles and in CBR at L=1000. 20 problems per datapoint. Y-axis is on a log scale.

of synchronous cycles, L is a factor indicating the relative communication speed, and CCC is the total concurrent constraint checks.

On some of the largest problems that we tested on, they did not terminate within our test time limit, which was set to 5 hours (in order to insure that experiments finished in a reasonable amount of time). This only occurred on a small number of cases, and in these cases we still collected cycle and CCC counts from the job. These counts are a lower bound on the runtime of the problem. Therefore they still provide an estimate of the job length, and at worst the true counts would be higher than the measured ones. In general this would only make our results stronger because the high runtimes from non-terminations generally occurred with reference heuristics (eg., random) which we have concluded are the least efficient (and hence they caused the long runtimes).

3.3.1 Performance of AdoptMVA versus Adopt

Our initial tests showed when we compared AdoptMVA to Adopt, with each one using their own agent ordering heuristics, the results were inconclusive. We therefore wanted to compare Adopt-MVA to Adopt on a level playing field, controlling for agent ordering so that any performance difference could not be attributed merely to the different ordering heuristics. We used a static lexicographic agent ordering for both algorithms, giving them exactly the same ordering on all problems. The intra-agent search heuristic used in AdoptMVA was the MVA-HigherVars heuristic.



(a) Cycles (b) CBR

Figure 3.3: Using equivalent lexicographic agent orderings, on graph coloring problems with 4 variables per agent and link density 2, AdoptMVA outperforms Adopt in terms of cycles and in CBR at L=1000. 20 problems per datapoint. Y-axis is on a log scale.

We found that when agent ordering is controlled for, AdoptMVA has a lower cycle count than Adopt in high density meeting scheduling and graph coloring problems (see Figs 3.2a and 3.3a). This result is consistent with our understanding of centralization, since in general, algorithms that use more centralization can terminate in a lower number of cycles (but with higher total constraint checks). Since AdoptMVA uses fewer communication cycles, it would perform better than Adopt in systems with a high communication cost (eg., L=1000 in Fig 3.2b). The difference in CBR was statistically significant ($p < 0.05$) for all problem sizes except one case. For meeting scheduling at # agents = 5, high variance on the larger problem size caused higher p values. We ran a larger dataset of 40 problems for that problem size which brought the p value down from 0.22 to 0.09. We believe a larger dataset of around 100 problems would reduce the p value to below 0.05.

We did not present results for AdoptMVA on low density meeting scheduling because the improvement in cycles for that class of problems is minor. With only 2 variables per agent, it is nearly the same as Adopt's 1 variable per agent. This is a factor to consider when choosing a DCOP algorithm; AdoptMVA is more beneficial when agents have more than 2 variables per agent.

AdoptMVA compared to OptAPO

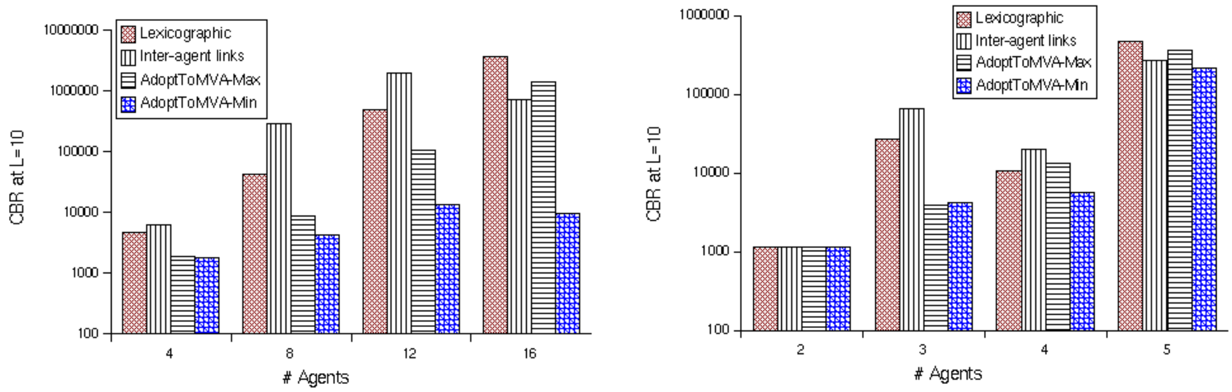
Note that our results with AdoptMVA are similar to our finding in Chapter 2 that OptAPO would be more efficient than Adopt at high communication costs (L=1000 or higher). AdoptMVA, like OptAPO, uses partial centralization to reduce the communication cycles. However, the important difference between these algorithms is that AdoptMVA only makes use of the centralization that already exists in the problem, while OptAPO centralizes variables external to an agent. Another difference is that the cost of local search in AdoptMVA is more controlled, because the local search will never be larger than the number of meetings owned by an agent. OptAPO, on the other hand, can centralize up to as many variables in the entire problem, which can be much more costly than AdoptMVA's smaller searches. If OptAPO could be modified in the future to restrict its centralizing action to only variables that are already available to an agent, it might be a viable alternative to AdoptMVA.

Agent ordering

In this experiment we used a lexicographic ordering for both Adopt and AdoptMVA and found that AdoptMVA had fewer cycles than Adopt. In other initial experiments where we used different agent ordering heuristics for each algorithm, it was inconclusive whether Adopt or AdoptMVA performed better. In some cases Adopt performed better, but on other cases AdoptMVA performed better.

This raised the question: does there exist a heuristic for AdoptMVA which on average will outperform Adopt? Our experiment in section 3.3.2 brought progress towards determining a good heuristic for AdoptMVA, but the heuristics there do not yet consistently outperform Adopt's best heuristic.

We hypothesize that the reduced granularity of the AdoptMVA agent ordering could prevent it from attaining an optimal ordering on par with the finer granularity Adopt ordering. An agent ordering inherently has a coarser granularity than the ordering that can be used in traditional Adopt. This was illustrated previously in Figure 3.1 which showed that Adopt has the ability to construct an ordering on the agents' variables, including the possibility of *interleaving* two agent's variables. The AdoptMVA ordering is more limited because it can only order agents. It is conceivable that this makes it impossible in some cases to capture the finer grain constraint relationships that Adopt can account for. However, it is also possible that further research on heuristics for AdoptMVA could reveal one that addresses this problem.



(a) low density (b) high density

Figure 3.4: Performance of four agent ordering heuristics on low density (a) and high density (b) meeting scheduling problems. Datapoints are based on the average of 20 problems.

3.3.2 Comparison of agent ordering heuristics

One of the important components of an Adopt based algorithm is the ordering of the agents. During initial tests of AdoptMVA we found the ordering had a large effect on performance.

To gain a better understanding of the heuristics (implemented according to the descriptions in section 3.2.4), we compared their performance on meeting scheduling and graph coloring problems. For the Branch & Bound intra-agent ordering, we used the MVA-HigherVars heuristic because the experiment discussed in the next section showed it to be the most efficient.

We compared four agent ordering heuristics, on low and high density meeting scheduling (Fig. 3.4), and graph coloring (Fig. 3.5). As expected, we saw that the agent ordering can make a tremendous difference - for some of the problems in Figure 3.4, the difference between two heuristics is one or two orders of magnitude in size.

As can be seen in Tables 3.2 and 3.3, AdoptToMVA-Min was the best performing heuristic on 8 out of the 11 problem sizes tested. This difference was significant in some of the cases. On the other cases, wide variance in the runtimes caused the p values to be greater than 0.05. From this we believe the AdoptToMVA-Min heuristic is a good agent ordering heuristic, though future work may find one that does better with less variability.

# Agents	Best Heuristic	Lexicographic	Inter-agent links	AdoptToMVA-Max	AdoptToMVA-Min
4	AdoptToMVA-Min	0.11	**0.05**	0.29	-
8	AdoptToMVA-Min	0.21	0.20	0.20	-
12	AdoptToMVA-Min	**0.04**	0.15	**0.03**	-
16	AdoptToMVA-Min	<**0.001**	0.14	**0.01**	-
2	none	-	-	-	-
3	AdoptToMVA-Max	0.26	**0.02**	-	0.10
4	AdoptToMVA-Min	0.09	0.09	0.31	-
5	AdoptToMVA-Min	0.20	0.72	0.21	-

Table 3.2: A list of the best performing agent ordering heuristics in Figure 3.4, with low density meeting scheduling at the top followed by high density meeting scheduling. The p value presented is the paired t-test value when comparing the best heuristic to each of the other heuristics. Bold text indicates statistically significant results ($p < 0.05$).

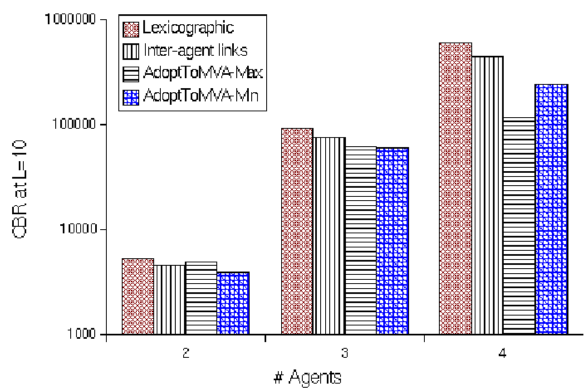


Figure 3.5: Performance of four agent ordering heuristics on graph coloring problems with 4 variables per agent. Datapoints are based on the average of 20 problems.

# Agents	Best Heuristic	Lexicographic	Inter-agent links	AdoptToMVA-Max	AdoptToMVA-Min
2	AdoptToMVA-Min	0.21	0.33	0.34	-
3	AdoptToMVA-Min	**0.04**	0.12	0.93	-
4	AdoptToMVA-Max	0.08	**0.02**	-	0.15

Table 3.3: A list of the best performing agent ordering heuristics on graph coloring problems from Figure 3.5. The p value presented is the paired t-test value when comparing the best heuristic to the next best heuristic.

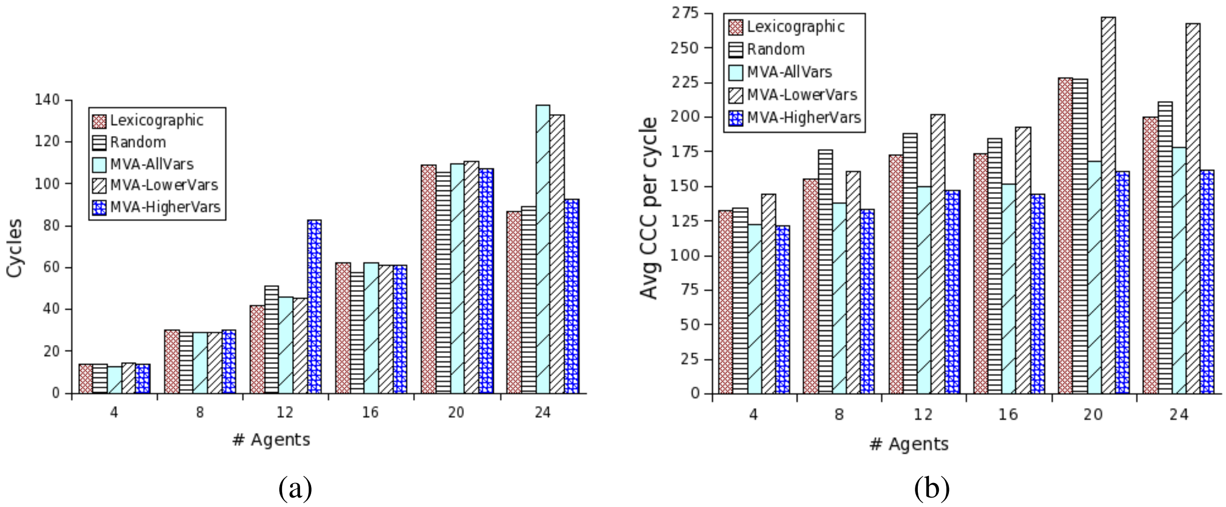


Figure 3.6: Intra-agent search heuristics on low density meeting scheduling (2 meetings per agent). 20 problems per datapoint.

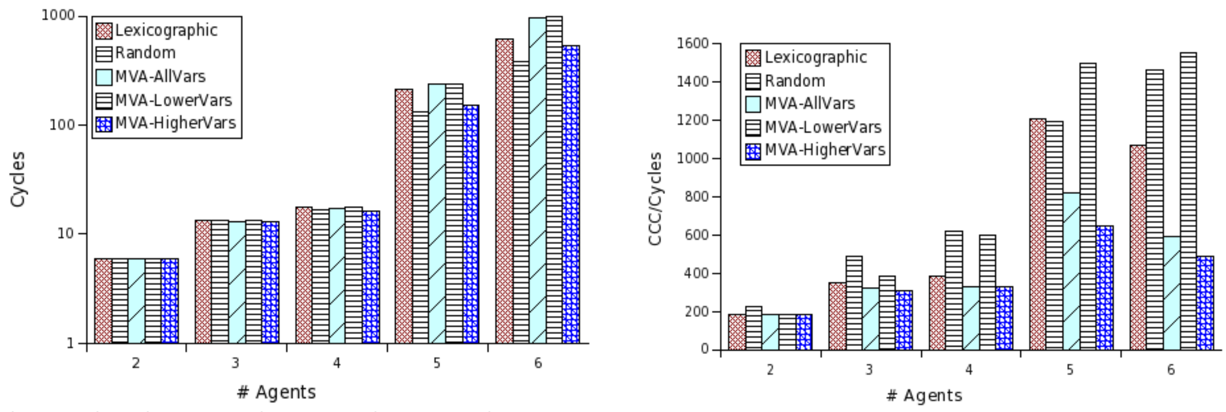
3.3.3 Comparison of intra-agent search heuristics

In order to reduce the number of constraint checks used by the AdoptMVA Branch & Bound search, we experimented with the intra-agent variable ordering heuristics described in section 3.2.3. For all of the tests, the inter-agent ordering was the AdoptToMVA-Min order since our previous experiment indicated it performed best more often than the other heuristics.

# Agents	Best Heuristic	Lexicographic	Random	MVA-AllVars	MVA-LowerVars	MVA-HigherVars
4	MVA-HigherVars	**0.03**	**0.002**	0.59	**0.005**	-
8	MVA-HigherVars	**0.005**	**0.01**	0.13	**0.001**	-
12	MVA-HigherVars	**0.005**	**<0.001**	0.44	**<0.001**	-
16	MVA-HigherVars	**<0.001**	**0.002**	0.13	**<0.001**	-
20	MVA-HigherVars	**0.02**	**<0.001**	**0.02**	**0.001**	-
24	MVA-HigherVars	**<0.001**	**<0.001**	0.13	0.055	-

Table 3.4: A list of the best performing intra-agent search ordering heuristics on low density meeting scheduling, according to the average CCC per cycle from Figure 3.6. The p values are for a paired t-test between the best heuristic and each of the other heuristics.

To determine the efficiency of each heuristic at improving the search pruning, we collected the total CCC. However, constraint checks are correlated with the number of total cycles (since more cycles will result in more constraint checks). Since our heuristics affected the number of cycles, we can compare their actual computational efficiency by using their average CCC per cycle ($TotalCCC/TotalCycles$).



(a) (b)

Figure 3.7: Intra-agent search heuristics on high density meeting scheduling (4 meetings per agent). 20 problems per datapoint.

Meeting Scheduling

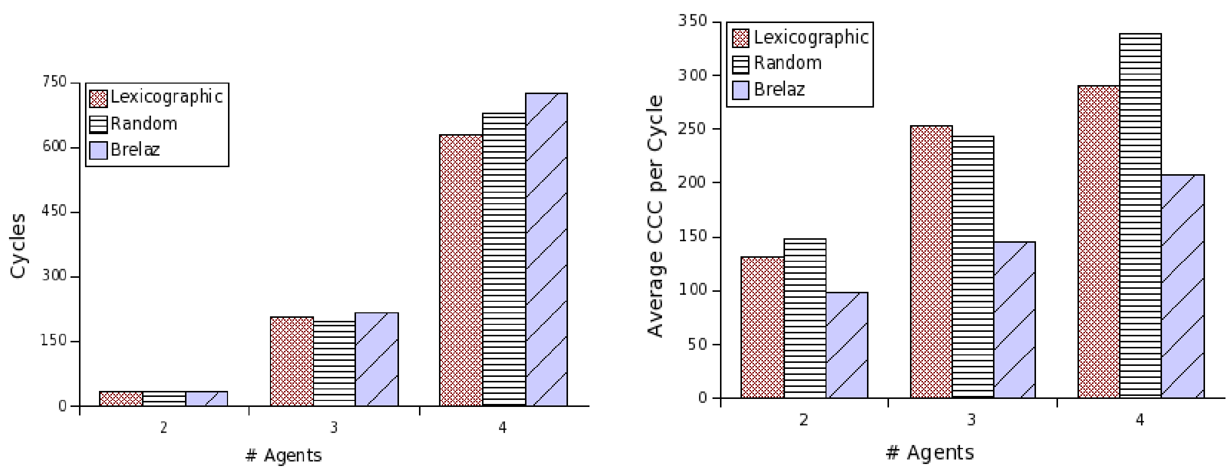
From Figures 3.6b and 3.7b, we see that MVA-HigherVars had the lowest average CCC per cycle in all cases. This difference is statistically significant in nearly all cases (see Tables 3.4 and 3.5), excluding comparison to MVA-AllVars. MVA-AllVars was frequently almost as efficient as MVA-HigherVars because MVA-AllVars is the combination of heuristics MVA-LowerVars and MVA-HigherVars. Therefore it is sometimes able to benefit from the influence of MVA-HigherVars. We can see that the MVA-LowerVars heuristic does not contribute useful information because it performed the worst in most cases. We can conclude that MVA-HigherVars is the best intra-agent search heuristic of the five we compared, because it has the lowest average computation per cycle.

Another interesting result this experiment produced is that the intra-agent heuristics slightly affected the number of cycles which it took the algorithm to terminate (see Fig 3.6a). Since the variable ordering within the Branch & Bound search only directly affects the search, not the external Adopt algorithm, one might expect that the ordering would only influence constraint checks, not cycles. However, the heuristics do in fact affect cycles because they change the optimal solutions which are produced by the search. There often may be several solutions that have the same cost, and a different variable ordering can enable us to find an alternative one that provides a faster path to the final solution.

# Agents	Best Heuristic	Lexicographic	Random	MVA-AllVars	MVA-LowerVars	MVA-HigherVars
2	MVA-HigherVars	1.0	**<0.001**	1.0	1.0	-
3	MVA-HigherVars	**0.001**	**0.005**	0.18	**<0.001**	-
4	MVA-HigherVars	**0.04**	**<0.001**	0.98	**0.01**	-
5	MVA-HigherVars	0.08	**<0.001**	0.44	**0.02**	-
6	MVA-HigherVars	**0.004**	**<0.001**	0.07	**<0.001**	-

Table 3.5: A list of the best performing intra-agent search ordering heuristics on high density meeting scheduling, according to the average CCC per cycle from Figure 3.7. The p values are for a paired t-test between the best heuristic and each of the other heuristics. The results for # Agents = 2 were not significant because most of the heuristics on such a small problem were identical.

From the high density meeting scheduling (Fig 3.7a) we see that the random heuristic took fewer cycles in several cases, particularly the problem sizes with agents = 5 and 6. Although random is an uninformed heuristic, the fact that it picks a different random order each cycle may help it to find the optimal solution faster simply because it got "lucky" and found a good solution path. Indeed, researchers have found that randomization heuristics sometimes outperform the best known informed heuristics [15].



(a) (b)

Figure 3.8: Intra-agent search heuristics on graph coloring problems. 20 problems per datapoint.

However, our random heuristic did not have the lowest constraint checks, and from it's high average CCC per cycle we can conclude that it is not efficient for minimizing computational cost. A interesting future task would be to attempt creating an informed heuristic with a small amount of randomness, which might maintain efficiency while also benefiting from the random exploration.

Graph Coloring

For graph coloring problems, we tested only with the lexicographic, random, and Brelaz heuristics, since the MVA heuristics are not intended for graph coloring.

From Figure 3.8b it is apparent that the Brelaz heuristic is the most efficient computationally, and this difference is statistically significant (Table 3.6). This confirms that the Brelaz heuristic, as we would expect, is a good heuristic for graph coloring problems with multiple variables per agent.

# Agents	Best Heuristic	Lexicographic	Random	Brelaz
2	Brelaz	**0.04**	**0.01**	-
3	Brelaz	**0.04**	**0.006**	-
4	Brelaz	**0.01**	**0.008**	-

Table 3.6: A list of the best performing intra-agent search ordering heuristics on graph coloring, according to the average CCC per cycle from Figure 3.8. The p values are for a paired t-test between the best heuristic and each of the other heuristics.

3.3.4 Meeting Scheduling as agents and meeting size are scaled

In order to gain a better understanding of meeting scheduling performance as certain properties are scaled, we also conducted experiments in which the number of attendees per meeting was varied. We constructed a set of meeting scheduling problems with 2 meetings per agent (M), and varied the number of agents (N) and the number of attendees per meeting (A). Increasing the number of attendees per meeting increases the link density of the problem. The dataset included 10 problems per problem size.

Figure 3.9a shows the CBR measurements for Adopt at 3 different values of A. Several of the initial points for Adopt at A=3 and A=4 are very high because one or two of the problems in each dataset were outliers. They took two orders of magnitude longer than the other problems in the same class, so we believe that the inter-agent ordering used in those cases was detrimental. When we remove the outliers from the dataset, we have Figure 3.9b which is easier to interpret. We define outliers as cases which were more than two standard deviations away from the mean.

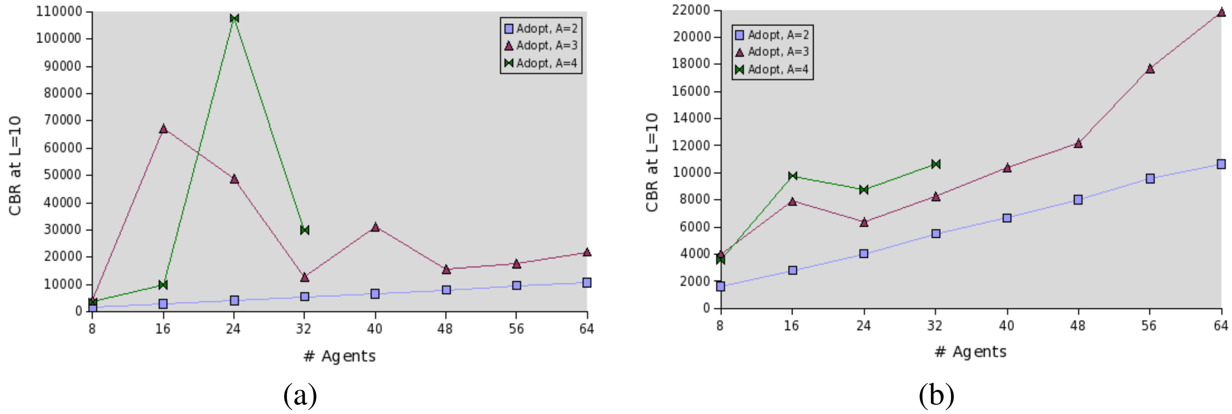


Figure 3.9: Meeting scheduling as number of agents and number of attendees per meeting (A) are increased. Adopt's CBR (at L=10) for three levels of A (2, 3, and 4) is shown. (a) shows the original results, and (b) shows the results after several outliers were removed. The size of the meetings appears to have a significant effect on performance.

From Figure 3.9b we see that increasing the number of attendees causes performance to become slower at all problem sizes. In many cases, increasing the size of the meetings detriments the runtime more than increasing the number of agents. For example, at # Agents = 32 and A=2, increasing the meeting size to A=4 has as much of a performance impact as if we had doubled the number of agents to 64. We therefore believe that meeting size is an important factor to consider when designing meeting scheduling problems, since it can have a surprisingly large effect on solution difficulty.

3.4 Related Work

A good deal of prior work has been done in building personal assistants [16][17][18]. Although most have not used fully complete optimization algorithms, they address many of the human-computer interface issues that arise. The Electric Elves [18] agent assistance technology identified the importance of "adjustable autonomy" - the need for agents to vary their level of autonomy, sometimes leaving important decisions to humans.

The CMRADAR project [19] is engaged in building tools for automatically scheduling meetings and determining the user interfaces that will work best for this application. It also aims to account for individual scheduling preferences (for example, one may prefer meetings in the afternoon).

Oh and Smith have developed methods for learning a user's time preferences for scheduling [20]. Their learning agents observe a user's decisions during scheduling, and use this information to construct an accurate statistical model of the user's preferences. Preferences such as these could be applied to the algorithms used in this thesis, which are modeled as a DCOP. Since a DCOP is based on a cost function which allows us to assign arbitrary costs to variables, it can easily model meeting scheduling situations in which meetings have different priorities (costs).

Modi and Veloso have investigated the effect of rescheduling, or bumping meetings, on the scheduling problem[12]. Bumping refers to the act of rescheduling a previously scheduled meeting in order to accommodate a new meeting. They found that an informed heuristic based on scheduling difficulty can reduce the total number of bumps used. The scheduling difficulty of an agent is computed from the density of its calendar, following from the intuition that participants with sparser schedules are easier to reschedule and therefore are preferred bump candidates.

An extension to the MAP model called Private, Incremental MAP (piMAP) [12] takes into account the incremental nature of meeting scheduling and the privacy concerns. The incremental aspect allows for new variables and constraints to be added to the problem over time. The privacy requirement says that any agent in $participants(V_i)$ can not communicate information about variable V_i to any agent who is not in $participants(V_i)$. In other words, the agents participating in a given meeting can not provide information about that meeting to agents who are not involved with the meeting.

Adopt improvements

Research on Adopt has found preprocessing techniques that provide agents with improved initial lower bounds that can speed up the algorithm by an order of magnitude [21]. The authors used dynamic programming techniques which reduced the number of partial solutions that Adopt generates and revisits in graph coloring and distributed sensor networks.

The Adopt algorithm has recently been applied by Maheswaran, et al. to the meeting scheduling domain, with two particularly notable results [3]:

- Meeting scheduling problems took much longer to solve than graph coloring problems with comparable number of variables and constraints. This may indicate that the applied problem of meeting scheduling is fundamentally different from the abstract graph coloring domain from a computational perspective.

- Preprocessing and runtime heuristic optimizations allowed an order of magnitude speedup.

The authors use a *passup* heuristic to precompute lower bounds estimates in a distributed manner, and thus decrease the amount of search that the Adopt execution needs to do. Each variable in the constraint tree performs local optimizations to determine a lower bound, and then passes up the estimate to its parent which can use it to make a more informed estimate of its own lower bound. They also use an improved tree hierarchy for the constraint graph. Since Adopt's normal DFS tree may not be a minimum depth tree, they use MLSP trees which are often shorter and experimentally were shown to speed up the Adopt algorithm.

The results we presented here do not include the authors' heuristics, but we use several other simple optimizations which were sufficient to support our conclusions. The precomputation of bounds and improved tree hierarchy could still be applied in future applications to gain larger improvements.

3.5 Conclusions

We have developed a modified algorithm called AdoptMVA for DCOP domains that have multiple variables per agent. When applied to meeting scheduling using a generic agent ordering, AdoptMVA completes in fewer cycles than Adopt, and has a lower Cycle-Based Runtime at high communication latencies. Although none of the agent orderings tested were decisively superior, future work may uncover a heuristic to address this.

We empirically determined a meeting scheduling Branch & Bound search heuristic that is statistically better than the other heuristics that were tested. This finding is supported by logically reasoning that higher priority variables, the ones the heuristic considers, have a more important impact on the local search. Finally, we have provided a snapshot of how the meeting scheduling problem scales as number of agents and meeting size are increased.

Chapter 4

Conclusions

This work has focused on the effect of centralization in DCOPs, and makes several contributions to our understanding in that area. The key conclusions we reached were:

- Algorithmic centralization can reduce communication cycles, but increases local computation costs, potentially making the algorithm more expensive than one that communicates more. We contributed the Cycle-Based Runtime (CBR) metric to aid in comparing algorithms that use differing amounts of centralization. Using CBR we found that the Adopt algorithm performs better than OptAPO on graph coloring problems, assuming reasonably inexpensive communication is available.

- Domain centralization, such as in meeting scheduling, naturally lends itself to algorithms that take advantage of local information. We outline an alternative algorithm called AdoptMVA which uses centralized search within agents to make better use of the problem structure. When we control for inter-agent ordering, AdoptMVA completes in fewer cycles than Adopt. We also develop a Branch & Bound search heuristic for meeting scheduling which empirically is the most efficient of the ones tested.

- Based on the results of our work, we believe that in order for partially centralized algorithms to be practical, we need ways of limiting the amount that is centralized. Otherwise, as was seen in Chapter 2 with OptAPO, the search problem can grow to large sizes which are very expensive to solve. AdoptMVA on the other hand limits its centralization to the number of variables that are already available within an agent and this provides a cap on the size of the local search. Future work involving DCOP centralization should continue to consider ways of limiting the size of the centralized search.

4.1 Future Work

Meeting scheduling, due to its exponential nature, will likely remain a challenging problem for some time. An important direction will lie in finding the right heuristics and algorithms to be able to solve very large scheduling problems in a reasonable time.

We believe improvements to AdoptMVA could be made by further researching the search heuristics:

- During our experiments with agent ordering heuristics for AdoptMVA, it was observed that certain heuristics worked well sometimes, but not all the time. While there was no clear superior heuristic, it might be possible to combine heuristics to obtain the best aspects of all of them. A future task would be to identify what features, if any, could be used to determine which heuristic to use.

- For the Branch & Bound search heuristics for meeting scheduling, it would be worth developing a heuristic that is both informed and randomly varied. This could help reduce communication cycles by increasing the chances of finding a fast solution path, while still keeping the computational costs efficient by using an informed ordering.

It would also be interesting to test DCOP algorithms in a fully distributed setting. This might confirm whether CBR holds up as a good representation of distributed runtime, and would provide an estimate of what communication overhead (L value) is realistic.

Bibliography

[1] Modi, P.J., Shen, W., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence Journal (2005)

[2] Yokoo, M.: Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems. Springer (2001)

[3] Maheswaran, R.T., Tambe, M., Bowring, E., Pearce, J.P., Varakantham, P.: Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In: AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, Washington, DC, USA, IEEE Computer Society (2004) 310–317

[4] Horling, B., Lesser, V., Vincent, R.: Multi-Agent System Simulation Framework. 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation (2000)

[5] Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. Knowledge and Data Engineering **10** (1998) 673–685

[6] Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing Performance of Distributed Constraints Processing Algorithms. In: Proc. Workshop on Distributed Constraint Reasoning (AAMAS). (2002)

[7] Modi, P.J.: Distributed Constraint Optimization for Multiagent Systems. PhD thesis, University of Southern California (2003)

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

[9] Freuder, E.C., Wallace, R.J.: Partial constraint satisfaction. Artif. Intell. **58** (1992) 21–70

[10] Mailler, R.: A Mediation-Based Approach to Cooperative, Distributed Problem Solving. PhD thesis, University of Massachussetts at Amherst (2004)

[11] Davin, J., Modi, P.: Impact of problem centralization in distributed constraint optimization algorithms. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). (2005)

[12] Modi, P., Veloso, M.: Bumping strategies for the multiagent agreement problem. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). (2005)

[13] Wallace, R.: Enhancements of branch and bound methods for the maximal constraint satisfaction problem. In Jampel, M., Freduer, E., Maher, M., eds.: OCS'95: Workshop on Over-Constrained Systems at CP'95, Cassis, Marseilles (1995)

[14] Brélaz, D.: New methods to color the vertices of a graph. Communications of the ACM **22** (1979) 251–256

[15] Cicirello, V.A., Smith, S.F.: Amplification of search performance through randomization of heuristics. In: CP '02: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, London, UK, Springer-Verlag (2002) 124–138

[16] Maes, P.: Agents that reduce work and information overload. Communications of the ACM **37** (1994)

[17] Mitchell, T.M., Caruana, R., Freitag, D., McDermott, J., Zabowski, D.: Experience with a learning personal assistant. Communications of the ACM **37** (1994) 80–91

[18] Chalupsky, H., Gil, Y., Knoblock, C., Lerman, K., Oh, J., Pynadath, D., Russ, T., e, M.T.: Electric elves: Applying agent technology to support human organizations. In: Proceedings of Innovative Applications of Artificial Intelligence Conference. (2001)

[19] Modi, P.J., Veloso, M., Smith, S., Oh, J.: Cmradar: A personal assistant agent for calendar management. In: Agent Oriented Information Systems, (AOIS). (2004)

[20] Oh, J., Smith, S.: Learning user preferences for distributed calendar scheduling. In: Proc. 5th International Conference on Practice and Theory of Automated Timetabling (PATAT), Pittsburgh, PA (2004)

[21] Syed Ali, Sven Koenig, M.T.: Preprocessing techniques for accelerating the dcop algorithm adopt. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). (2005)