A CLASS OF ALGORITHMS FOR DISTRIBUTED CONSTRAINT OPTIMIZATION

THÈSE N^{O} 3942 (2007)

PRÉSENTÉE LE 30 OCTOBRE 2007

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire d'intelligence artificielle

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Adrian PETCU

ingénieur diplômé de l'Université Polytechnique de Bucarest, Roumanie
et de nationalité roumaine

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. B. Faltings, directeur de thèse
Prof. D. Parkes, rapporteur
Prof. M. A. Shokrollahi, rapporteur
Prof. M. Yokoo, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2007

To my family

Abstract

Multi Agent Systems (MAS) have recently attracted a lot of interest because of their ability to model many real life scenarios where information and control are distributed among a set of different agents. Practical applications include planning, scheduling, distributed control, resource allocation, etc. A major challenge in such systems is coordinating agent decisions, such that a globally optimal outcome is achieved. Distributed Constraint Optimization Problems (DCOP) are a framework that recently emerged as one of the most successful approaches to coordination in MAS.

This thesis addresses three major issues that arise in DCOP: efficient optimization algorithms, dynamic and open environments, and manipulations from self-interested users. We make significant contributions in all these directions: *Efficiency-wise*, we introduce a series of DCOP algorithms, which are based on dynamic programming, and largely outperform previous DCOP algorithms. The basis of this class of algorithms is DPOP, a distributed algorithm that requires only a linear number of messages, thus incurring low networking overhead. For *dynamic environments* we introduce self-stabilizing algorithms that can deal with changes and continuously update their solutions. For *self interested users*, we propose the M-DPOP algorithm, which is the first DCOP algorithm that makes *honest behaviour* an ex-post Nash equilibrium by implementing the VCG mechanism *distributedly*. We also discuss the issue of *budget balance*, and introduce two algorithms that allow for *redistributing* (some of) the VCG payments back to the agents, thus avoiding the welfare loss caused by wasting the VCG taxes.

Keywords: artificial intelligence, constraint optimization, dynamic systems, multiagent systems, self-interest

Résumé

Les systèmes multiagent (MAS) ont récemment attiré beaucoup d'intérêt en raison de leur capacité de modéliser beaucoup de scénarios réels où l'information et le contrôle sont distribués parmi un ensemble de différents agents. Les applications pratiques incluent la planification, l'ordonnancement, les systèmes de contrôle distribués, ou encore l'attribution de ressources. Un défi important dans de tels systèmes est la coordination des décisions des agents, afin que des résultats globalement optimaux soient obtenus. Les problèmes d'optimisation distribuée sous contraintes (DCOP) sont un cadre qui a récemment émergé comme étant une des approches les plus performantes pour la coordination de MAS.

Cette thèse adresse trois points principaux de DCOP : les algorithmes efficaces d'optimisation, les environnements dynamiques et ouverts, et les manipulations par des agents stratégiques. Nous apportons des contributions significatives dans toutes ces directions : en ce qui concerne l'*éfficacité*, nous présentons une série d'algorithmes de DCOP qui sont basés sur la programmation dynamique, et offrent des performances considerablement meilleures que les algorithmes précédents. La base de cette classe d'algorithmes est DPOP, un algorithme distribué qui exige seulement un nombre linéaire de messages, économisant ainsi des ressources de réseau. Pour les *environnements dynamiques*, nous présentons des algorithmes auto-stabilisants qui peuvent prendre en compte des changements dans l'environnement et mettent à jour les solutions en temps réel. Pour *agents stratégiques*, nous proposons l'algorithme *M-DPOP*, qui est le premier algorithme de DCOP qui fait du *comportement honnête* un équilibre post-Nash en appliquant le mécanisme de VCG *de façon distribuée*. Nous discutons également de la question de l*équilibre du budget*, et présentons deux algorithmes qui permettent de *redistribuer* [partiellement] les paiements VCG aux agents, évitant ainsi la perte d'utilité provoquée par le gaspillage des taxes VCG.

Mots-clés : intelligence artificielle, optimisation sous contraintes, systèmes dynamiques, systèmes multiagent, agents stratégiques

Acknowledgements

Boi Faltings, my adviser, has kindly offered me the chance to work at the LIA, at a time when I had already packed my luggage to go somewhere else. He offered me high quality guidance throughout these years, saw the big picture when I could not, and pushed me back on track when I was diverging to other “cute little problems”. He offered a good working environment and plenty of resources such that I can concentrate on research, and a lot of freedom to organize my work. Thank you!

I appreciate the time and effort offered by the members of my jury: Makoto Yokoo, David Parkes, Rachid Guerraoui, Amin Shokrollahi.

Makoto Yokoo laid the foundations of distributed constraint satisfaction in the early nineties, and brought significant contributions ever since. Without him, I would have had to write a thesis about something surely less interesting.

I appreciate very much the ongoing collaboration with David Parkes since last year. He provided invaluable help with M-DPOP and BB-M-DPOP’s nice game theoretic formalization, analysis, proofs, rigorous reviews, and great feedback in general.

Rina Dechter kindly accepted to help with the formalization of DFS-based algorithms into the AND/OR framework, with the formalization of several aspects of Dynamic DCOP and self-stabilization, and with clarifying the relationship between distributed and centralized algorithms.

Following, there is a list of people who have helped me in various ways throughout this thesis. I hope my failing memory has not forgotten too many of them: Dejan Kostic, Thomas Léauté, Radoslaw Szymanek, Daniel Tralamazza have provided useful input for devising the “Network Based Runtime” metric. Marius Silaghi: privacy guru, initial introduction to DisCSPs. Awe-inspiring remark in Bologna. Roger Mailler: help with the experimental evaluation of PC-DPOP against OptAPO. Akshat Kumar: working out the details of the implementation of CDDs for H-DPOP, lots of patience while running experiments, and interesting insights on the comparison between H-DPOP and search algorithms. Wei Xue: working out many intricate details of the implementation of BB-M-DPOP, interesting insights, and lots of patience with running experiments, even by remote from China (incredibly responsive, too) Thomas Leaute: thanks for our nice collaboration, and for the help with the French

abstract. Ion Constantinescu: a very good matchmaker. Without him, I would have done a PhD somewhere else (I don't regret I haven't). Useful advice about PhD and life in general. Good luck on US soil, Ion! Steven Willmott: useful coaching in the beginning. Radu Jurca: Interesting and fruitful discussions throughout, an idea with online O-DPOP, and latex issues. George Ushakov: working on a meeting scheduling prototype, and re-discovering the maximum-cardinality-set heuristic for low-width DFS trees while working on his Master's project. Great job! Aaron Bernstein: discussions and useful thoughts about the DFS reconstruction in the early days of M-DPOP. Jeffrey Shneidman: useful feedback on an early version of the M-DPOP paper, and the idea of using the operator placement as an example application. Giro Cavallo: interesting discussions on incentives and redistribution schemes during my visit to Harvard, and making me feel welcome together with Florin Constantin.

My colleagues from LIA, for good feedback on research issues and presentational skills, or simply for providing good atmosphere during coffee breaks: Marita Ailomaa, Arda Alp, Walter Binder, Ion Constantinescu, Jean-Cedric Chappelier, Carlos Eisenberg. Radu Jurca, Thomas Léauté, Santiago Macho-Gonzalez, Nicoleta Neagu, David Portabela, Cecilia Perez, Jamila Sam-Haroud, Michael Schumacher, Vincent Schickel-Zuber, Radoslaw Szymanek, Paolo Viappiani, Steven Willmott. A special thank you to Marie Decrauzat, who was always helpful with personal and administrative issues.

My friends from Lausanne and far away (apologies to the ones I forgot, they're equally dear to me): Vlad &Ilinca Badilita, Vlad &Cristina Bucurescu, Irina Ceaparu, Vlad and Mari Chirila, Nicolae Chiurtu, Ion Constantinescu, Razvan Cristescu, Serban Draganescu, Adrian and Daniela Gheorghe, Petre Ghitescu, Peter and Laura Henzi, Adrian Ionita, Vlad Lepadatu, Nicoleta Neagu, Camil and Camelia Petrescu, Mugur Stanciu (helped me make my mind about doing a PhD).

Thanks to my parents, my sister, and to my brother in law, who were very supportive during these past few years. Last but certainly not least, the biggest thank you goes to my children for the sunshine they have brought to my life, and especially to my wife, Nicoleta, for bearing with me (or rather with my absence) for so long. Thank you for allowing me to pursue this dream, and for teaching me life's most important lesson.

"*Do, or do not. There is no* try."

— JMY

Contents

Chapter 1

Introduction

"A journey of a thousand miles begins with a single step."
— Lao tzu

Many real problems are naturally distributed among a set of *agents*, each one holding its own subproblem. Agents are *autonomous* in the sense that they have control over their own subproblems, and can choose their actions freely. They are *intelligent*, in the sense that they can reason about the state of the world, the possible consequences of their actions, and the utility they would extract from each possible outcome. They may be *self-interested*, i.e. they seek to maximize their own welfare, regardless of the overall welfare of their peers. Furthermore, they can have *privacy* concerns, in that they may be willing to cooperate to find a good solution for everyone, but they are reluctant to divulge private, sensitive information.

Examples of such scenarios abound. For instance, producing complex goods like cars or airplanes involves complex supply chains that consist of many different actors (suppliers, sub-contractors, transport companies, dealers, etc). The whole process is composed of many subproblems (procurement, scheduling production, assembling parts, delivery, etc) that can be globally optimized all at once, by expressing everything as a constraint optimization problem. Another quite common example is meeting scheduling ([127, 141, 239]), where the goal is to arrange a set of meetings between a number of participants such that no meetings that share a participant are overlapping. Each participant has preferences over possible schedules, and the objective is to find a feasible solution that best satisfies everyone's preferences.

Traditionally, such problems were solved in a centralized fashion: all the subproblems were communicated to one entity, and a centralized algorithm was applied in order to find the optimal solution. In contrast, a distributed solution process does not require the centralization of all the problem in a single location. The agents involved in the problem preserve their autonomy and the control over their local

problems. They will communicate via messages with their peers in order to reach agreement about what is the best joint decision which maximizes their overall utility. Centralized algorithms have the advantage that they are usually easier to implement, and often faster than distributed ones. However, centralized optimization algorithms are often unsuitable for a number of reasons, which we will discuss in the following.

Unboundedness: it may be unpractical or even impossible to gather the whole problem into a single place. For example, in meeting scheduling, each agent has a (typically small) number of meetings within a rather restricted circle of acquaintances. Each one of these meetings possibly conflicts with other meetings, either of the agent itself, or with meetings of its partners. When solving such a problem in a centralized fashion, it is not known a priory which ones of these potential conflicts will manifest themselves during a solving process. Therefore, it is required that the centralized solver acquire all the variables and constraints of the whole problem beforehand, and apply a centralized algorithm in order to guarantee a feasible (and optimal) solution. However, in general it is very difficult to bound the problem, as there is always another meeting that involves one more agent, which has another meeting, and so on. This is a setting where distributed algorithms are well suited, because they do not require the centralization of the whole problem in a single place; rather, they make small, local changes, which eventually lead to a conflict-free solution.

Privacy: is an important concern in many domains. For example, in the meeting scheduling scenario, participating in a certain meeting may be a secret that an agent may not want to reveal to other agents not involved in that specific meeting. Centralizing the whole problem in a solver would reveal all this private information to the solver, thus making it susceptible to attacks, bribery, etc. In contrast, in a distributed solution, usually information is not leaked more than required for the solving process itself. Learning everyone's constraints and valuations becomes much more difficult for an attacker.

Complex Local Problems: each agent may have a highly complex local optimization problem, which interacts with (some of) its peers' subproblems. In such settings, the cost of the centralization itself may well outweigh the gains in speed that can be expected when using a centralized solver. When centralizing, each agent has to formulate its constraints on all imaginable options beforehand. In some cases, this requires a huge effort to evaluate and plan for all these scenarios; for example, a part supplier would have to precompute and send all combinations of delivery dates, prices and quantities of many different types of products it is manufacturing.

Latency: in a dynamic environment, agents may come in the system or leave at all times, change their preferences, introduce new tasks, consume resources, etc. If such a problem is solved centrally, then the centralized solver should be informed of all the changes, re-compute solutions for each change,

and then re-distribute the results back to the agents. In case changes happen fast, the latency introduced by this lengthy process could make it unsuitable for practical applications. In contrast, a distributed solution where small, localized changes are dealt with using local adjustments can potentially scale much better and adapt much faster to changes in the environment.

Performance Bottleneck: when solving the problem in a centralized fashion, all agents sit idle waiting for the results to come from the central server, which has to have all the computational resources (CPU power, memory) to solve the problem. This makes the central server a performance bottleneck. In contrast, a distributed solution better utilizes the computational power available to each agent in the system, which could lead to better performance.

Robustness: to failures is a concern when using a single, centralized server for the whole process, which is a single point of failure. This server may go offline for a variety of reasons (power or processor failure, connectivity problems, DOS attacks, etc). In such cases the entire process is disrupted, whereas in a distributed solution, the fact that a single agent goes offline only impacts a small number of other agents in its vicinity.

All these issues suggest that in some settings, distributed algorithms are in fact the only viable alternative. To enable distributed solutions, agents must communicate with each other to find an optimal solution to the overall problem while each one of them has access to only a part of this problem.

Distributed Constraint Satisfaction (DisCSP) is an elegant formalism developed to address constraint satisfaction problems under various distributed models assumptions[38, 39, 203, 205, 225]. When solutions have degrees of quality, or cost, the problem becomes an optimization one and can be phrased as a Constraint Optimization Problem or COP[189]. Indeed the last few years have seen increased research focusing on the more general framework of distributed COP, or DCOP[81, 141, 160, 237].

Informally, in both the DisCSP and the DCOP frameworks, the problem is expressed as a set of individual subproblems, each owned by a different agent. Each agent's subproblem is connected with some of the neighboring agents' subproblems via constraints over shared variables. As in the centralized case, the goal is to find a globally optimal solution. But now, the computation model is restricted. The problem is distributed among the agents, which can release information only through message exchange among agents that share relevant information, according to a specified algorithm.

Centralized CSP and COP are a mature research area, with many efficient techniques developed over the past three decades. Compared to the centralized CSP, DisCSP is still in its infancy, and thus current DCOP algorithms typically seek to adapt and extend their centralized counterparts to distributed environments. However, it is very important to note that the performance measures for distributed algorithms are radically different from the ones that apply to centralized one. Specifically, if in centralized optimization the *computation time* is the main bottleneck, in distributed optimization it is rather the

communication which is the limiting factor. Indeed, in most scenarios, message passing is orders of magnitude slower than local computation. Therefore it becomes apparent that it is desirable to design algorithms that require a minimal amount of communication for finding the optimal solution. This important difference makes designing efficient *distributed* optimization algorithms a non-trivial task, and one cannot simply hope that a simple distributed adaptation of a successful centralized algorithm will work as efficiently.

1.1 Overview

This thesis is organized as follows:

Part I: Preliminaries and Background: Chapter 2 introduces the DCOP problem, and a set of definitions, notations and conventions. Chapter 3 overviews related work and the current state of the art.

Part II: The DPOP Algorithm: Chapter 4 introduces the dynamic programming *DPOP* algorithm. Chapter 5 introduces the H-DPOP algorithm, which shows how consistency techniques from search can be exploited in DPOP to reduce message size. This is a technique that is orthogonal to most of the following algorithms, and can therefore be applied in combination with them as well.

Part III: Tradeoffs: This part discusses *extensions* to the DPOP algorithm which offer different tradeoffs for difficult problems. Chapter 6 introduces MB-DPOP, an algorithm which provides a customizable tradeoff between Memory/Message Size on one hand, and Number of Messages on the other hand. Chapter 7 discusses two algorithms (A-DPOP and LS-DPOP) that trade optimality for reductions in memory and communication requirements. Chapter 8 discusses an alternative approach to difficult problems, which centralizes high width subproblems and solves them in a centralized way.

Part IV: Dynamics: This part discusses distributed problem solving in *dynamic environments*, i.e. problems can change at runtime. Chapter 9 introduces two self-stabilizing algorithms that can operate in dynamic, distributed environments. Chapter 10 discusses *solution stability* in dynamic environments, and introduces a self-stabilizing version of DPOP that maintains it.

Part V: Incentives: In this part we turn to systems with self-interested agents. Chapter 11 discusses systems with *self-interested users*, and introduces the M-DPOP algorithm, which is the first distributed algorithm that ensures *honest behaviour* in such a setting. Chapter 12 discusses the issue of *budget*

balance, and introduces two algorithms that extend M-DPOP in that they allow for *redistributing* (some of) the VCG payments back to the agents, thus avoiding the welfare loss caused by wasting the taxes.

Finally, Chapter 13 presents an overview of the main contributions of this thesis in Section 13.1, and then concludes.

Part I

Preliminaries and Background

Chapter 2

Distributed Constraint Optimization Problems

"United we can't be, divided we stand."

—

This chapter introduces the Distributed Constraint Optimization formalism (Section 2.1), a set of assumptions we make for the most part of this thesis (Section 2.2), and a number of applications of DCOP techniques (Section 2.3).

2.1 Definitions and Notation

We start this section by introducing the centralized Constraint Optimization Problem (COP)[19, 189]. Formally,

Definition 1 (COP) *A discrete* constraint optimization problem *(COP) is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ *such that:*

- $\mathcal{X} = \{X_1, ..., X_n\}$ *is a set of* variables *(e.g. start times of meetings);*
- $\mathcal{D} = \{d_1, ..., d_n\}$ *is a set of discrete, finite variable* domains *(e.g. time slots);*
- $\mathcal{R} = \{r_1, ..., r_m\}$ *is a set of* utility functions*, where each* r_i *is a function with the scope* $(X_{i_1}, \cdots, X_{i_k})$*,* $r_i : d_{i_1} \times .. \times d_{i_k} \rightarrow R$*. Such a function assigns a* utility (reward) *to each possible combination of values of the variables in the scope of the function. Negative amounts mean* costs*. Hard constraints (which forbid certain value combinations) are a special case of*

utility functions, which assign 0 to feasible tuples, and $-\infty$ to infeasible ones; [1]

The goal is to find a complete instantiation $\mathcal{X}^*$ for the variables X_i that *maximizes* the sum of utilities of individual utility functions. Formally,

$$X^* = argmax_X \left(\sum_{r_i \in \mathcal{R}} r_i(X) \right) \tag{2.1}$$

where the values of r_i are their corresponding values for the particular instantiation X. The *constraint graph* is a graph which has a node for each variable $X_i \in \mathcal{X}$ and a hyper-edge for each relation $r_i \in \mathcal{R}$.

Using Definition 1 of a COP, we define the Constraint Satisfaction Problem as a special case of a COP:

Definition 2 (CSP) *A discrete* constraint satisfaction problem *(CSP) is a COP $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ such that all relations $r_i \in \mathcal{R}$ are hard constraints.*

Remark 1 (Solving CSPs) *CSPs can obviously be solved with algorithms designed for optimization: the algorithm has to search for the solution of minimal cost (which is 0, if the problem is satisfiable).*

Definition 3 (DCOP) *A discrete* distributed constraint optimization problem *(DCOP) is a tuple of the following form: $\langle \mathcal{A}, \mathcal{COP}, \mathcal{R}^{ia} \rangle$ such that:*

- *$\mathcal{A} = \{A_1, \ldots, A_k\}$ is a set of agents (e.g. people participating in meetings);*
- *$\mathcal{COP} = \{COP_1, \ldots COP_k\}$ is a set of disjoint, centralized COPs (see Def. 1); each COP_i is called the* local subproblem of agent *A_i, and is owned and controlled by agent A_i;*
- *$\mathcal{R}^{ia} = \{r_1, \ldots r_n\}$ is a set of* interagent utility functions *defined over variables from several different local subproblems COP_i. Each $r_i : scope(r_i) \rightarrow R$ expresses the rewards obtained by the agents involved in r_i for some joint decision. The agents involved in r_i have full knowledge of r_i and are called "responsible" for r_i. As in a COP, hard constraints are simulated by utility functions which assign 0 to feasible tuples, and $-\infty$ to infeasible ones;*

Informally, a DCOP is thus a multiagent instance of a COP, where each agent holds its own local subproblem. Only the owner agent has full knowledge and control of its local variables and constraints. Local subproblems owned by different agents can be connected by *interagent utility functions $\mathcal{R}^{ia}$* that specify the utility that the involved agents extract from some joint decision. *Interagent hard constraints*

[1] Maximizing the sum of all valuations in the constraint network will choose a feasible solution, if one exists.

that forbid or enforce some combinations of decisions can be simulated as in a COP by utility functions which assign 0 to feasible tuples, and $-\infty$ to infeasible ones. The interagent hard constraints are typically used to model domain-specific knowledge like "a resource can be allocated just once", or "we need to agree on a start time for a meeting". It is assumed that the interagent utility functions are known to all involved agents.

We call the *interface variables* of agent A_i the subset of variables $\mathcal{X}_i^{ext} \subseteq \mathcal{X}_i$ of COP_i, which are connected via interagent relations to variables of other agents. The other variables of A_i, $\mathcal{X}_i^{int} \subset \mathcal{X}_i$ are called *internal variables*, and are only visible to A_i. We have that $\mathcal{X}_i = \mathcal{X}_i^{int} \sqcup \mathcal{X}_i^{ext}$.

As in centralized COP, we define the *constraint graph* as the graph which is obtained by connecting all the *variables* which share a utility function. We call *neighbors* two agents which share an interagent utility function. The *interaction graph* is the graph which is obtained by connecting pairwise all the *agents* which are neighbors. Subsequently, we will assume that only agents which are connected in the interaction graph are able to communicate directly.

As in the centralized case, the task is to find the optimal solution to the *COP* problem. In traditional centralized COP, we try to have algorithms that minimize the running time. In DCOP, the algorithm performance measure is not just the time, but also the communication load, most commonly the number of messages.

As for centralized CSPs, we can use Definition 3 of a DCOP to define the Distributed Constraint Satisfaction Problem as a special case of a DCOP:

Definition 4 (DisCSP) *A discrete* distributed *constraint satisfaction problem (DisCSP) is a DCOP* $< \mathcal{A}, \mathcal{COP}, \mathcal{R}^{ia} >$ *such that (a)* $\forall COP_i \in \mathcal{COP}$ *is a CSP (all internal relations are hard constraints) and (b) all* $r_i \in \mathcal{R}^{ia}$ *are hard constraints as well.*

Remark 2 (Solving DisCSPs) *DisCSPs can obviously be solved with algorithms designed for DCOP: the algorithm has to search for the solution of minimal cost (which is 0, if the problem is satisfiable).*

Remark 3 (DCOP is NP-hard)

2.2 Assumptions and Conventions

In the following we present a list of assumptions and conventions we will use throughout the rest of this thesis.

2.2.1 Ownership and control

Definition 3 states that each agent A_i owns and controls its own local subproblem, COP_i. To simplify the exposition of the algorithms, we will use a common simplifying assumption introduced by Yokoo et al. [225]. Specifically, we represent the whole COP_i (and agent A_i as well) by a single tuple-valued meta variable X_i, which takes as values the whole set of combinations of values of the interface variables of A_i. This is appropriate since all other agents only have knowledge of these interface variables, and not of the internal variables of A_i.

Therefore, in the following, we denote by "agent" either the physical entity owning the local subproblem, or the corresponding meta-variable, and we use "agent" and "variable" interchangeably.

2.2.2 Identification and communication patterns

Theoretical results (Collin, Dechter and Katz[38]) show that in the absence of agent identification (i.e. in a network of uniform nodes), even simple constraint satisfaction in a ring network is not possible. Therefore, in this work, we assume that each agent has an unique ID, and that it knows the IDs of its neighbors.

We further assume that neighboring agents that share a constraint know each other, and can exchange messages. However, agents that are not connected by constraints are not able to communicate directly. This assumption is realistic because of e.g. limited connectivity in wireless environments, privacy concerns, overhead of VPN tunneling, security policies of some companies may simply forbid it, etc.

2.2.3 Privacy and Self-Interest

For the most part of this thesis (Part 1 up to and including Part 4), we assume that *the agents are not self-interested* i.e. each one of them seeks to maximize the overall sum of utility of the system as a whole. Agents are expected to work cooperatively towards finding the best solution to the optimization problem, by following the steps the algorithm as presscribed. Furthermore, *privacy is not a concern*, i.e. all constraints and utility functions are known to the agents involved in them. Notice that this does not mean that an agent not involved in a certain constraint has to know about its content, or even its existence.

In Part 5 we relax the assumption that the agents are cooperative, and discuss systems with self-interested agents in Chapters 11 and 12.

2.3 Example Applications

There is a large class of multiagent coordination problems that can be modeled in the DCOP framework. Examples include distributed timetabling problems[104], satellite constellations[11], multiagent teamwork[206], decentralized job-shop scheduling[205], human-agent organizations[30], sensor networks[14], operator placement problems in decentralized peer-to-peer networks[71, 173], etc. In the following, we will present in detail a multiagent meeting scheduling application example[127, 171, 239].

2.3.1 Distributed Meeting Scheduling

Consider a large organization with dozens of departments, spread across dozens of sites, and employing tens of thousands of people. Employees from different sites/departments (these are the agents $\mathcal{A}$) have to set up hundreds/thousands of meetings. Due to all the reasons cited in the introduction, a centralized approach to finding the best schedule is not desirable. The organization as a whole desires to minimize the cost of the whole process (alternatively, maximize the sum of the individual utilities of each agent) [2].

Definition 5 (Meeting scheduling problem) *A* meeting scheduling problem *(MSP) is a tuple of the following form:* $\langle \mathcal{A}, \mathcal{M}, \mathcal{P}, \mathcal{T}, \mathcal{C}, \mathcal{R} \rangle$ *such that:*

- $\mathcal{A} = \{A_1, ..., A_k\}$ *is a set of agents;*
- $\mathcal{M} = \{M_1, ..., M_n\}$ *is a set of meetings*
- $\mathcal{P} = \{p_1, ..., p_k\}$ *is a set of mappings from agents to meetings: each* $p_i \subseteq \mathcal{M}$ *represents the set of meetings that* A_i *attends;*
- $\mathcal{T} = \{t_1, ..., t_n\}$ *is a set of time slots: each meeting can be held in one of the available time slots;*
- $\mathcal{R} = \{r_1, ..., r_k\}$ *is a set of utility functions; a function* $r_i : p_i \rightarrow \mathbb{R}$ *expressed by an agent* A_i *represents* A_i*'s utility for each possible schedule of its meetings;*

In addition, we have hard constraints: two meetings that share a participant must not overlap, and the agents participating in a meeting must agree on a time slot for the meeting.

The goal of the optimization is to find the schedule which (a) is feasible (i.e. respects all constraints) and (b) maximizes the sum of the agents' utilities.

Proposition 1 *MSP is NP-hard.*

[2] A similar problem, called *Course Scheduling* is presented in Zhang and Mackworth[239].

Example 1 (Distributed Meeting Scheduling) *Consider an example where 3 agents want to find the optimal schedule for 3 meetings:* $A_1 : \{M_1, M_3\}$, $A_2 : \{M_1, M_2, M_3\}$ *and* $A_3 : \{M_2, M_3\}$. *There are 3 possible time slots to organize these three meetings:* $8AM, 9AM, 10AM$. *Each agent* A_i *has a local scheduling problem* COP_i *composed of:*

- *variables* $A_i\_M_j$*: one variable* $A_i\_M_j$ *for each meeting* M_j *the agents wants to participate in;*
- *domains: the 3 possible time slots:* $8AM, 9AM, 10AM$*;*
- *hard constraints which impose that no two of its meetings may overlap*
- *utility functions: model agent's preferences*

Figure 2.1 shows how this problem is modeled as a DCOP. Each agent has its own local subproblem, and Figure 2.1(a) shows COP_1*, the local subproblem of* A_1. COP_1 *consists of 2 variables* $A_1\_M_1$ *and* $A_1\_M_3$ *for* M_1 *and* M_3*, the meetings* A_1 *is interested in.* A_1 *prefers to hold meeting* M_1 *as late as possible, and models this with* r_1^0 *by assigning high utilities to later time slots for* M_1. A_1 *cannot participate both in* M_1 *and in* M_3 *at the same time, and models this with* r_1^1 *by assigning* $-\infty$ *to the combinations which assign the same time slot to* M_1 *and* M_3*. Furthermore,* A_1 *prefers to hold meeting* M_3 after M_1*, and thus assigns utility 0 to combinations in the upper triangle of* r_1^1*, and positive utilities to combinations in the lower triangle of* r_1^1.

To ensure that the agents agree on the time slot allocated for each meeting, they must coordinate the assignments of variables in their local subproblems. To this end, we introduce inter-agent equality constraints between variables which correspond to the same meeting. Such a constraint associates utility 0 with combinations which assign the same value to the variables involved, and $-\infty$ for different assignments. In Figure 2.1(b) we show each agent's local subproblem, and interagent equality constraints which connect corresponding variables from different local subproblems. For example, c_1 models the fact that A_1 and A_2 must agree on the time slot which will be allocated to M_1. This model of a meeting scheduling problem as a DCOP corresponds to the model in[127].

This distributed model of the meeting scheduling problem allows each agent to decide on its own meeting schedule, without having to defer to a central authority. Furthermore, the model also preserves the *autonomy* of each agent, in that an agent can choose not to set its variables according to the specified protocol. Assuming this is the case, then the other agents can decide to follow his decision, or hold the meeting without him.

2.3.2 Distributed Combinatorial Auctions

Auctions are a popular way to allocate resources or tasks to agents in a multiagent system. Essentially, bidders express bids for obtaining a good (getting a task in reverse auctions). Usually, the highest

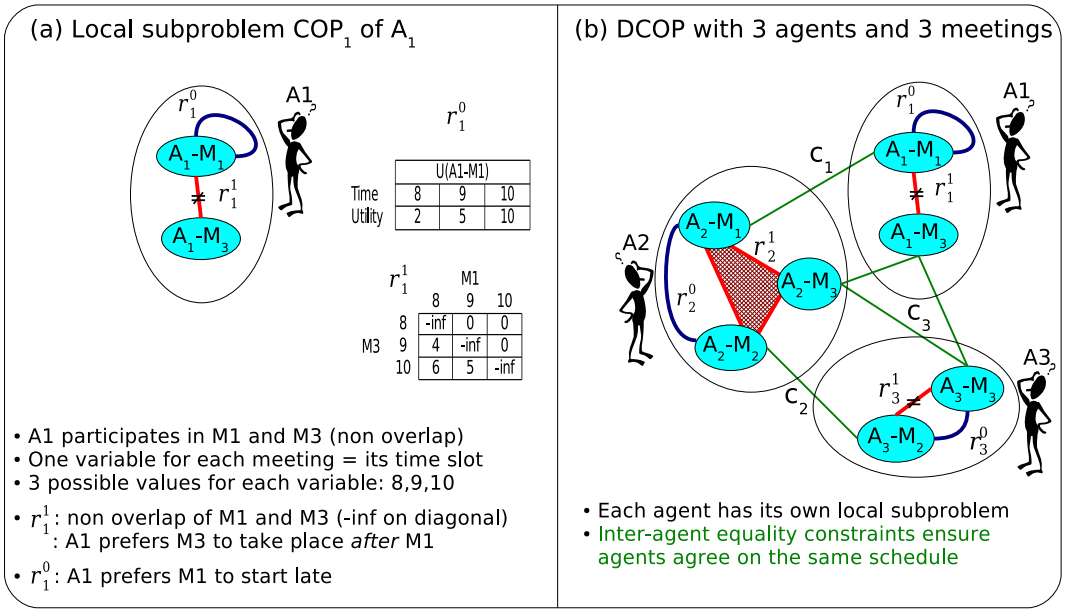


Figure 2.1: *A meeting scheduling example. (a) is the local subproblem of agent A_1 (each meeting has an associated variable that models its allocated time slot; r_1^1 models the non-overlap of M_1 and M_3, and the fact that A_1 prefers to have M_3 after M_1; r_1^0 expresses A_1's preference to have M_1 as late as possible;) (b) DCOP model where agreement among agents is enforced with inter-agent equality constraints c_1, c_2, c_3.*

bidder (lowest one in reverse auctions) gets the good (task in reverse auctions), for a price that is either his bid (first price auctions) or the second highest bid (second price, or Vickrey auctions).

Combinatorial auctions (CA) are much more expressive because they allow bidders to express bids on bundles of goods (tasks), thus being most useful when goods are complementary or substitutable (valuation for the bundle does not equal the sum of valuations of individual goods).

CAs have received a lot of attention for a few decades now, and there is a large body of work dealing with CAs that we are not able to cover here (a good survey appears in[47]). There are many applications of CAs in multiagent systems like resource allocation[148], task allocation[217], etc.

There are also many available algorithms for solving the allocation problem (e.g. CABOB[185]). However, most of them are centralized: they assume an auctioneer that collects the bids, and solves the problem with a centralized optimization method.

There are few non-centralized methods for solving CAs. Fujita et al. propose in[80] a *parallel* branch and bound algorithm for CAs. The scheme does not deal with incentives at all, and works by splitting the search space among multiple agents, *for efficiency reasons*. Narumanchi and Vidal propose in[145] several distributed algorithms, some suboptimal, and an optimal one, but which is computationally expensive (exponential in the number of agents).

Definition 6 (Combinatorial Auction) *A* combinatorial auction *(CA) is a tuple* $< \mathcal{A}, \mathcal{G}, \mathcal{B} >$ *such that:*

- $\mathcal{A} = \{A_1, ..., A_k\}$ *is a set of bidding agents;*
- $\mathcal{G} = \{g_1, ..., g_n\}$ *is a set of (indivisible) goods*
- $\mathcal{B} = \{b_1, ..., b_k\}$ *is a set of bids; a bid* b_k^i *expressed by an agent* A_i *is a tuple* $\langle A_i, G_k^i, v_k^i \rangle$*, where* v_k^i *is the valuation agent* A_i *has for the bundle of goods* $G_k^i \subseteq \mathcal{G}$*; when* A_i *does not obtain the whole set* G_k^i*, then* $v_k^i = 0$*;*

A *feasible allocation* is a mapping $S : \mathcal{B} \rightarrow \{$ `true,false` $\}$ that assigns `true` or `false` to all bids b_k^i (`true` means that agent A_i wins its bid b_k^i) such that $\forall b_k^i, b_l^m$, if $\exists g_j \in \mathcal{G}$ s.t. $g_j \in G_k^i \wedge g_j \in G_l^m$ (where G_k^i and G_l^m are sets of goods comprised in the two bids b_k^i and b_l^m, respectively), then at least one of b_k^i, b_l^m is assigned `false`. In words, no two bids that share a good can both win at the same time (because goods are assumed to be indivisible). The value of an allocation S is $val(S) = \sum_{b_k^i \in \mathcal{B} s.t. S(b_k^i)=true} v_k^i$

Proposition 2 *Finding the optimal allocation* $S^* = argmax_S(val(S))$ *is NP-hard[182] and inapproximable[183].*

We detail in the following how to cast CAs to a DCOP model. Let us assume that agent A_i has bid $b_i = \langle A_i, G_i, v_i \rangle$. For each good $g_j \in G_i$, A_i creates a local variable g_j^i that models the winner of good g_j. The domain of this variable is composed of the agents interested in good g_j (the ones whose bids contain g_j).

A_i connects all variables g_j^i from its local problem with a relation r_i that assigns v_i only to the combination of values $\langle g_j^i = A_i, g_k^i = A_i, \dots \rangle$ (the one that assigns all goods $g_j^i \in b_i$ to A_i), and 0 to all other combinations.

Example 2 (Distributed Combinatorial Auctions) *See Figure 2.2 for an example CA with 3 bidders and 3 goods. Figure 2.2(a) shows a centralized constraint optimization model of the problem. The variables represent goods, and each one has as possible values the agents which bid on that good. Assigning a variable* g_k *to one of its values* A_i *means that* A_i *will get good* g_k*. The relations expressed by agents on subsets of variables model bids. For example, the bid* $b_1 = \langle A_1, \{g_1, g_3\}, 10 \rangle$ *of agent* A_1 *is modeled as the binary relation involving* g_1 *and* g_3 *in Figure 2.2(a). This relation assigns value 10 to the tuple* $\langle g_1 = A_1, g_3 = A_1 \rangle$*, and 0 to all other tuples* $\langle g_1 \times g_3 \rangle$*.*

Moving to a decentralized, DCOP model is shown in Figure 2.2(b). This involves each agent creating copies of the variables g_i *from the centralized model, and expressing their bids locally, as relations on the copy variables, just as in the centralized case. To ensure the feasibility of the resulting*

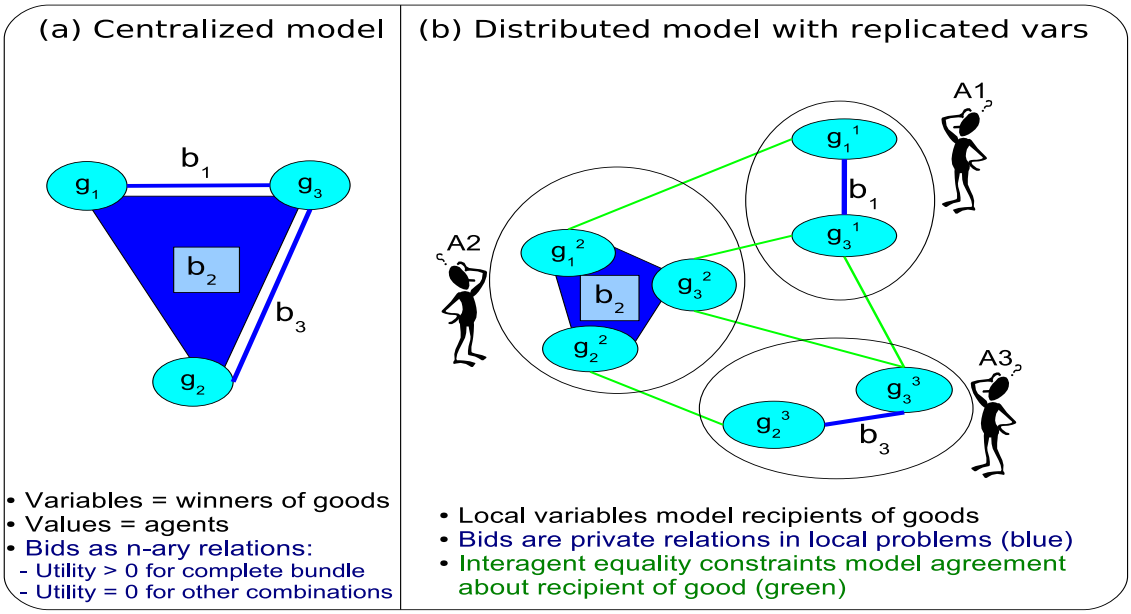


Figure 2.2: *Combinatorial Auctions modeled as Constraint Optimization. (a) shows a CA with 3 bidders and 3 goods modeled as a centralized COP, and (b) shows the equivalent decentralized DCOP model.*

allocation, we also need to connect all the copies that correspond to each good via equality constraints; thus, agreement about the final recipient of each good is ensured. For example, both agents A_1 and A_2 have bids on g_1. Therefore, they create local copies of the variable g_1, and connect these copies via the equality constraint as shown in Figure 2.2(b).

2.3.3 Overlay Network Optimization

Another setting for distributed constraint optimization is the optimal placement of data aggregation and processing operators on an overlay network[71, 100, 173]. In this application, there are multiple users and multiple servers. Each user is associated with a query and has a client machine located at a particular node on an overlay network. A query has an associated set of *data producers*, known to the user and located at nodes on the network. Each query also requires a set of data aggregation and processing operators, which should be placed on server nodes between the nodes with data producers and the user's node. Each user assigns a utility to different assignments of operators to servers to represent her preferences for different kinds of data aggregation. Examples of in-network operators for data aggregation include database style "join" operators, or custom logic provided by an end user. For instance, one may have an operator (snippet of code) that does database JOINs. Then, a user may desire "volcano data X" and "earthquake data Y" joined and sent to them. To address this, a specific operator that we call "VolcanoXEarthquakeY_Join" is created and put into the network. Naturally, each

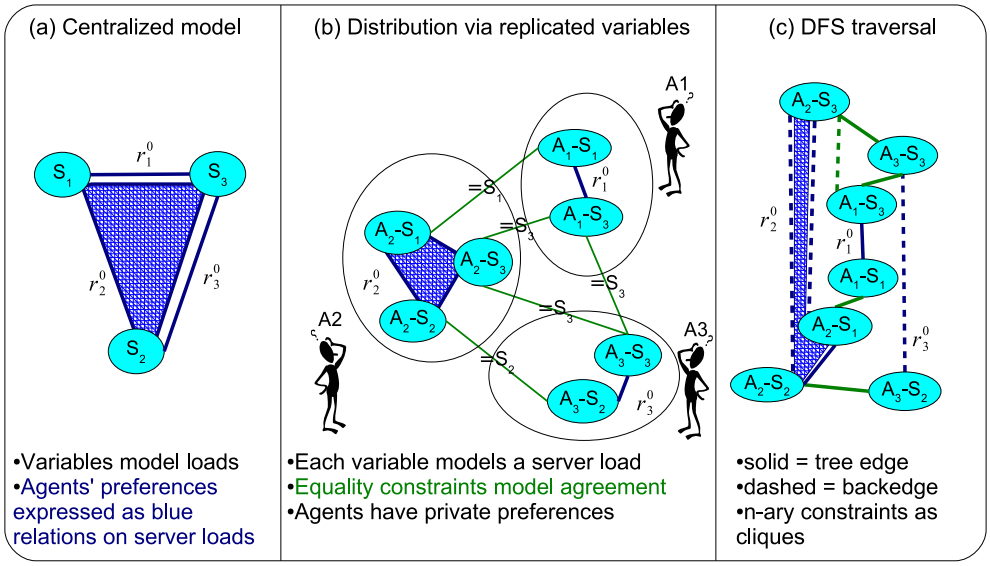


Figure 2.3: *An operator assignment problem.* ***(a)*** *The centralized COP model: each server has an associated variable that models the feasible combinations of operators that can be executed at the server's node. Agent preferences on assignments of operators are expressed as relations (blue); e.g.,* r_3^0 *states that* A_3 *obtains utility 10 if operator* $A_3\_o_1$ *runs on* S_2*, and only 5 if it runs on* S_3*.* ***(b)*** *The DCOP model with replicated variables, where agreement among agents is enforced with equality constraints between local copies;* ***(c)*** *A DFS arrangement of the graph in (b), used by the DPOP algorithm.*

user has preferences on the possible placement of their operators on servers in the network. The task is to allocate operators to servers such that capacity constraints are observed, and that the sum of utilities of individual users is maximized.

A distributed algorithm, to be executed by user clients situated on network nodes, will determine the assignment of data aggregation and processing operators to server nodes. The server nodes are assumed to "opt-in" in that they will implement whatever allocation is determined by users. Constraints on server nodes, e.g. based on maximal load, are commonly known to users and thus captured through public constraints. There are also other side-constraints because the queries have prerequisites that have to run on a server in order for the query to be executed there. Server nodes play no active role in the algorithm.

This problem maps easily to a DCOP model. Each user has a number of operators it would like to place. Each operator could be placed on (potentially) many servers, subject to the capacity constraints of the servers. To model this we introduce a variable for each server, which models in its domain the feasible combinations of operators that can be executed by that server. Each user has preferences on the possible placements of its operators.

Example 3 (Operator Placement) *In Figure 2.3, assume that* A_3 *wants to place an operator,* $A_3\_o_1$*. It has two alternatives: either on* S_2*, or on* S_3*. This is modeled as follows:* A_3 *has a variable for*

the feasible assignment of operators on servers S_2 and S_3. The domains of these variables contain all feasible combinations of operators each server can execute. Among these combinations, there are some that include A_3's operator, $A_3\_o_1$. Assume A_3 obtains utility 10 if its operator $A_3\_o_1$ runs on S_2, and only 5 if it runs on S_3. A_3 models this preference with the relation r_3^0, which assigns utility 10 to all cases in which S_2 runs its operator, 5 to all cases where S_3 runs its operator, 0 to all cases where its operator is not run anywhere, and $-\infty$ for cases where both S_2 and S_3 run the operator.

2.3.4 Distributed Resource Allocation

Definition 7 (Distributed sensor allocation problem (SAP)) *The distributed sensor network problem formalized in[14] consists of:*

- *a sensor field composed of n sensors: $S = \{s_1, s_2, ..., s_n\}$*
- *m targets that need to be tracked: $T = \{t_1, t_2, ..., t_m\}$*

Each sensor has a certain "range" (the maximum distance that it can cover), and in order to successfully track a target, 3 sensors have to be assigned to that target (triangulation can be applied using the data coming from those 3 sensors). The following restrictions apply:

- *any one-sensor can only track one target at a time;*
- *the sensors in the field can communicate among themselves, but not necessarily every sensor with every other sensor (the sensor connectivity graph is not fully connected). The 3 sensors tracking a given target must be able to communicate among themselves;*

We can formalize the problem as a DisCSP assigning one agent for each target: the variables are the required sensors (three variables per agent), and the values of each variable are the sensors that can track that target (are within range). Assume we have one agent A_i for each target T_i to be tracked. This agent would then have 3 variables to control: s_1^i, s_2^i, s_3^i; each of them represents one sensor that has to be assigned to track this target. The domain of the variables of each agent consists of sensors that can actually "see" the respective target).

In this representation of the problem, we have two types of constraints:

- intra-agent constraints (the constraints within one agent): (a) no two variables can be assigned the same value (one agent must have three *different* sensors tracking it) and (b) there must be a communication link between every two sensors that are assigned to each agent

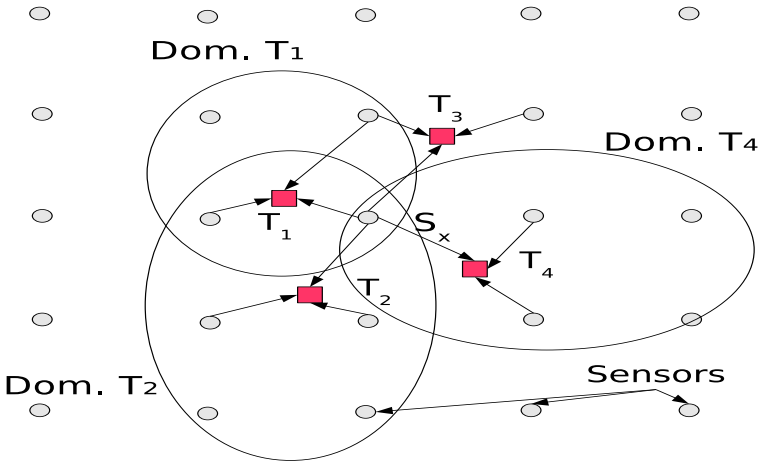


Figure 2.4: A sensor allocation problem example. 3 different sensors have to be allocated for each target. The figure shows allocation conflicts, as S_x is allocated to several targets at once.

- inter-agent constraints (the constraints between agents): no two variables s_k^i and s_l^j from any two agents A_i and A_j can be assigned the same value (one sensor can track a single target at a given time)

The problem is to allocate sensort to targets such that the maximal number of targets are tracked. Alternatively, each target can yield a "reward" for being tracked, and then the problem is to maximize the sum of rewards.

Proposition 3 *SAP is NP-hard.*

It is interesting to note that all constraints in this problem (except for the "visibility" ones) are constraints of mutual exclusion (typical in resource allocation problems).

Example 4 *Please refer to Figure 2.4 for an example SAP. The sensors are placed in a grid (filled circles) and the targest are scattered randomly in the grid (filled squares). The ovals depicted in the figure are each a* domain *of one of the targets (for example, $Dom.T_2$ contains all sensors that are within range of T_2). An arrow connecting a sensor to a target denotes that the sensor is allocated to that target. In the figure there are some conflicts, as the sensor S_x is allocated to multiple targets at the same time: T_1, T_2, T_3, T_4.*

2.3.5 Takeoff and Landing Slot Allocation

In this example, airports allocate takeoff and landing slots to different airlines and need to coordinate these allocations so that airlines have corresponding slots for their flights. Here, the airports and airlines

are agents; airports decide which airlines to allocate available slots to, while airlines decide which flights to operate. These decisions must be coordinated so that for every flight the airline has the required slots for its takeoff and landing. Nevertheless, airports want to keep control over the decision process as to which airline is allocate which ones of their available slots. Therefore, a centralized controller that would jointly optimize the whole slot allocation for all airports in the world is completely unrealistic.

Chapter 3

Background

Centralized Constraint Satisfaction Problems[48, 125, 142, 210] (CSP, see Definition 2) have been an area of active research since the 70's, when they were formalized for several applications including scene labeling in image processing[142]. CSPs have been extended to Constraint Optimization Problems[76, 79] (COP, see Definition 1) for problems which have solutions with different degrees of optimality or cost.

Algorithms for centralized CSP can be classified into two main categories of *search* (e.g., depth-first or best-first search[24, 85, 112, 215]) and *inference* (e.g., dynamic programming[15, 16, 19], variable elimination[50], join-tree clustering[51, 55]). Search algorithms have been enhanced with various forms of consistency techniques[21, 45, 56, 142], and with variations of the branch and bound principle[58] for optimization problems. Dynamic programming algorithms on the other hand have also been extended to bounded-error approximations, and also hybridized with search[106, 107, 119, 120, 180].

Search and inference algorithms can be distinguished primarily by their time and space complexities. An inference algorithm such as bucket-elimination[50] is time and space exponential only in the graph's treewidth. On the other hand, brute force search can operate with only linear memory but seems to lack structure-based time bounds, thus usually being time exponential in the size of the problem. Recently however, AND/OR search schemes were shown to accommodate graph-based bounds as well[133]. Specifically, AND/OR search spaces[146] for COPs and CSPs capture problem decomposition through AND nodes and they can be traversed in linear space and in time exponential in the depth of a spanning pseudo-tree of the problem's graph[54]. When caching of subproblem solutions is used[8, 42, 132], time and space complexity of those AND/OR search algorithms can be reduced to exponential in the treewidth as well.

In the early nineties, distributed constraint satisfaction was formalized[38, 205, 224, 225], and a first generation of distributed algorithms for DisCSP was proposed[223, 225, 226].

Naturally, emerging DCOP algorithms extend traditional centralized COP algorithms, and as such

fall into the two main categories of search and dynamic programming. We present in the following a comprehensive view of distributed CSP and COP algorithms. We show side by side search and inference algorithms, and discuss their strengths and weaknesses. In Section 3.1.1.2 we introduce new synchronous distributed AND/OR algorithms for COP having linear-size messages whose number is bounded exponentially in either the depth of the guiding DFS tree or in its treewidth, depending on the level of caching. Focusing on distributed inference in Section 3.2, we review the bucket-tree-elimination algorithm (BTE)[50, 107].

The strengths and weaknesses of distributed search and distributed inference are discussed and compared empirically throughout Parts II and III of this dissertation.

The focus of this thesis is on algorithms based on DFS structures, which we introduce and discuss in Section 3.4. However, a large body of work in the DisCSP arena is on algorithms that use arbitrary orderings, not necessarily DFS ones. For constraint *satisfaction*, the most prominent algorithms are the Asynchronous Backtracking (ABT) algorithm of Yokoo et al[225], the Asynchronous Weak Commitment search (AWC) of Yokoo[223], and the Asynchronous Search with Aggregations (AAS) by Silaghi et al.[197]. For constraint *optimization*, there is the Asynchronous Forward Bounding algorithm of Gershman et al[139]. We review all these algorithms in Section 3.1.

Parallel Constraint Satisfaction In a different line of research, Zhang and Mackworth[239] describe algorithms based on junction trees and tree decompositions for *parallel* constraint satisfaction/optimization. These algorithms are developed for problems which are initially centralized, and they assume that nodes from the junction tree can be assigned at will to agents to perform the respective computation, *for efficiency reasons*. In contrast, we are concerned with solving problems which are *distributed by nature*, and our algorithms seek to maintain the initial partition of the problem to the owner agents for several reasons, privacy included.

3.1 Backtrack Search in DCOP Algorithms

With few exceptions, the vast majority of work in distributed optimization has revolved around extending various forms of backtrack search[24, 85, 112, 215] originally designed for centralized COP, to a distributed environment. Loosely, centralized search algorithms work by establishing an ordering of the variables, and then executing a form of backtrack search based on that ordering. This works by assigning to variables values that are compatible with the values chosen for their ancestors, then moving forward to the next variable. When for a variable there is no value that is compatible with the values chosen for its ancestors, a backtrack occurs. The culprit assignment of its ancestors is called a *nogood*. For satisfaction algorithms, the search continues until either (a) an empty nogood is discovered (i.e. there is no solution to the problem) or (b) a full instantiation is discovered which does not contain any conflicts. For optimization algorithms, the search continues until ”enough“ of the search space is

explored to be able to infer that the optimal solution is found.

To increase efficiency, various schemes were developed which try to minimize the portion of the search space which has to be visited in order to prove that the algorithm has already found the optimal solution. The most well known such scheme is the *branch-and-bound* scheme from centralized optimization[58]. Branch and bound works as follows: as soon as we have a complete instantiation, we store it as the current best solution, and the cost of this instantiation as an *upper bound* on the cost that the algorithm tolerates. Later on during search, whenever a new value is tried for a variable, one computes the partial cost accumulated up to that variable, plus the cost incurred by the new instantiation. If this cost is larger than the current upper bound, then the assignment is *pruned*, as it cannot lead to a better solution than the current best solution, and the search backtracks. Whenever we find a new complete instantiation which has a lower cost than the current best solution, we update the best current solution to the new one, and the upper bound to the cost of this new solution.

DCOP algorithms typically seek to adapt and extend their centralized counterparts to distributed environments, and are based on the same principles: backtrack search, and some bounding scheme for pruning. However, it is very important to note that the performance measures for distributed algorithms are radically different from the ones that apply to centralized one. Specifically, if in centralized optimization the *computation time* is the main bottleneck, in distributed optimization it is rather the *communication* which is the limiting factor. Indeed, in most scenarios, message passing is orders of magnitude slower than local computation. This important difference makes designing efficient *distributed* optimization algorithms a non-trivial task of simply adapting centralized algorithms to work distributedly.

Execution Model DCOP algorithms are distinguished to be either synchronous or asynchronous. In the following, we describe briefly these two execution models in an informal way. In a *synchronous* algorithm, each agent waits for the messages it is supposed to receive from its peers, and only after having received them, it starts performing computation and sending out its own messages. In an *asynchronous* algorithm, all agents start performing computation and sending out messages even before having received any message from its peers. As incoming messages arrive, they incorporate them into their computation, and if needed, they send out updated messages of their own. The asynchronous execution has the potential advantage that agents don't sit idle waiting for messages, when they could possibly perform computations. On the other hand, a synchronous execution model ensures that agents perform their computation based on *relevant, most up to date* information. Therefore, the need to perform another computation when an updated message arrives is eliminated.

3.1.1 Synchronous search algorithms

In this section we discuss the SynchBB algorithm of Hirayama and Yokoo, and two synchronous algorithms that we have developed, and which work on a DFS tree.

3.1.1.1 Synchronous Branch and Bound (SynchBB)

The SynchBB algorithm is the first complete algorithm for DCOP, and was first proposed by Hirayama and Yokoo in[98]. This algorithm is a simple, distributed version of the classical centralized branch & bound scheme[118]. SynchBB does not use a DFS tree, but rather a linear ordering of the agents[1].

After an ordering is established (e.g. lexicographic ordering), the agents perform a synchronous branch and bound search. The process works like the centralized branch and bound algorithm; however, the agents, each associated with a single variable do not have access to the global upper/lower bounds on solution quality. This problem is addressed by simply passing these bounds back and forth, together with the forward value assignment messages and the backward backtrack messages.

This algorithm may require that any 2 agents/variables can communicate directly, thus violating our assumption from Section 2.2.2 which allows only for direct neighbors to communicate. Furthermore, it has been shown by Modi et al[141] to be quite inefficient, and is easily outperformed by more elaborate schemes.

Next, we will introduce a synchronous algorithm that performs an AND/OR search in a distributed fashion (Section 3.1.1.2), its branch and bound variant (Section 3.1.1.3), and we will also present the NCBB algorithm of Chechetka and Sycara (Section 3.1.2.3).

3.1.1.2 dAO-Opt: Simple AND/OR Search For DCOP

The AND/OR search spaces are a powerful concept for search that has been introduced by Nilsson in[146], and subsequently further developed in many other contexts[39, 78, 131, 135]. Recently, Dechter and Mateescu[54] showed how AND/OR graphs can capture search spaces for general graphical models that include constraint networks and belief networks. These AND/OR graphs are defined relative to the pseudo-tree of the graphical model.

AND/OR search spaces are a formalization of the idea that search on a pseudotree structure is potentially exponentially better than traditional search on linear variable orderings, *especially if caching is not allowed*. The reason is that when performed on a DFS structure (or more generally, on a pseudotree), search can be done in parallel on distinct branches of the tree. This yields search processes that in the worst case are time exponential in the depth of the tree. In contrast, traditional search algorithms

[1] One can think of this algorithm as working on a pseudochain, rather than on a pseudotree

that operate on linear variable orderings are time exponential in the number of variables. Therefore, it is always beneficial to perform search on DFS trees with low depth as opposed to linear variable orderings.

To see an example of this idea, consider the tree in Figure 3.3. It is easy to see that once X_0 is instantiated to a value from its domain, what remains is actually a set of two distinct subproblems, independent from each other. Therefore, they can be explored in parallel. The process can be applied recursively (instantiating X_1 as well leaves us with 2 independent subproblems - $\{X_3, X_7, X_8\}$ and $\{X_4, X_9, X_{10}\}$, which depend only on X_0 and X_1, but not on each other). On this particular example, the worst case complexity is reduced thus from $O(exp(14))$ (the depth of a linear ordering) to $O(exp(4))$ (the depth of the DFS tree).

Freuder[78], Bayardo and Miranker[13] and recently Dechter and Mateescu[54] describe search algorithms that apply this principle in a centralized setting. In[39] a distributed algorithm for constraint *satisfaction* is described. This algorithm also traverses an AND/OR search space, for finding a single solution. In the following, we introduce *dAO-Opt*: a *simple, synchronized* extension of AND/OR search for distributed optimization problems. As with[39], *dAO-Opt* also performs distributed search on a DFS tree in a depth-first manner, with the difference that it works for optimization problems as well. The formal description of *dAO-Opt* is presented in Algorithm 1.

Again, we start with a pre-established DFS tree. The root X_r starts the search process by assigning itself a value v_r^0 from its domain, and informing its children about this choice with an EVAL($\langle X_r = v_r^0 \rangle$) message. Each one of the children then picks a value for its variable, passes it down to its children, and so on. Each *EVAL* message sent to a child X_j of an agent X_i contains an assignment $\langle Sep_j \rangle$ for each variable in Sep_j, in order to allow X_j to evaluate the constraints it has with all its ancestors (not just with its parent).

When an agent X_i receives an $EVAL(\langle Sep_i \rangle)$ message from its parent, the message includes a full assignment of all variables in Sep_i. Given this assignment, X_i can evaluate those utility functions it has with its ancestors which are fully instantiated, for each one of its values $v_i^j \in dom(X_i)$. In the case of non-binary functions, X_i limits this evaluation to only the functions *in the bucket of* X_i[50], i.e. those relations whose scope does not include any of X_i's descendants; these functions are already fully instantiated, and can be evaluated by X_i.[2] The corresponding costs are denoted by $local\_cost(v_i^j, \langle Sep_i \rangle), \forall v_i^j \in dom(X_i)$:

Definition 8 (Local Cost) [3] *For each agent X_i, we denote by $local\_cost(v_i^j, \langle Sep_i \rangle) = \sum_{r_i \in X_i} \Big(r_i(v_i^j, \langle Sep_i \rangle) \Big)$, such that r_i is fully instantiated. This is the cost of its utility functions and constraints with its ancestors, when these ancestors are assigned the values as in $\langle Sep_i \rangle$, and $X_i = v_i^j$. If the assignment $X_i = v_i^j$ violates any such constraint, then the cost is infinite: $local\_cost(v_i^j, \langle Sep_i \rangle) = \infty$.*

[2] The functions which include in their scope X_i and descendants of X_i will be evaluated by X_i's descendant that is lowest in the DFS tree.

[3] The local cost as defined here is also called the *label* of the node in[54]

Algorithm 1 *dAO-opt - distributed AO search for cost minimization.*

dAO-opt($\mathcal{X}, \mathcal{D}, \mathcal{R}$): each agent X_i does:

Construct DFS tree: execute Alg 3; after completion, X_i knows P_i, PP_i, C_i, PC_i

procedure EVAL: X_i waits for EVAL($\langle Sep_i \rangle$) messages from P_i (unless root)

1 when EVAL($\langle Sep_i \rangle$) message received: let $\langle Sep_i \rangle$ be the assignment of Sep_i
2 **forall** $v_i \in dom(X_i)$ **do**
3 $\quad cost(v_i) \leftarrow local\_cost(v_i, \langle Sep_i \rangle)$
4 $\quad$ **forall** $X_j \in C_i$ **do**
5 $\quad\quad$ send EVAL($\langle Sep_{X_j} \rangle$) message to X_j containing the current $\langle Sep_{X_j} \rangle$
6 $\quad\quad$ wait for the COST message reply
7 $\quad\quad$ $cost(v_i) = cost(v_i) + cost_{X_j}(v_i)$
8 pick v_i^* s.t. $v_i^* = argmin_{v_i \in dom(X_i)}(cost(v_i))$
9 **if** X_i *is root* **then** v_i^* is the root's value in the optimal solution, and $cost(v_i^*)$ is the optimal cost
10 **else** send $COST(cost(v_i^*))$ to P_i

When *EVAL* messages have reached the leaves, or in case of a deadend (i.e. when an agent cannot find a value in its domain which is consistent with the assignments of its ancestors), the backtrack process begins. The leaves will cycle through their values, determine the best ones for the current instantiation of their ancestors, and reply with the best cost. A dead-ended agent replies with an infinite cost. Subsequently, whenever an agent X_i has received cost messages from all of its children for its current value, it tries the next: it informs the children about its new value assignment, and awaits the cost replies. When all its values are tried, the agent chooses the best one (minimal cost, or maximal utility, depending on whether we do minimization or maximization). The agent then reports the corresponding cost to its parent via a *COST* message[4], and the parent starts cycling through its values, and so on.

When the root X_r has cycled through all its values, and has received *COST* messages for each one, it can pick the best one. The cost (utility) associated with the root's best value is the optimal cost (utility) for the whole problem.

Re-deriving solutions for subtrees – extra work in the absence of caching: So far, this process allows only for determining the cost (utility) of the best solution, but not necessarily the solution itself. The reason is that this simple scheme does not do any caching. Therefore, when the root finds out what is its optimal value, and announces it, its children do not know what were their corresponding optimal values, and they will have to re-derive them. Therefore, there is another top-down search phase initiated by the root, where each agent announces its optimal value, and its children solve again their subtrees in the context of the values taken by their ancestors. Thus, smaller and smaller subtrees are solved again, for the purpose of re-deriving the optimal values of the roots of these subtrees, in the

[4]This cost is called the *value* of a node in[54]; we use the term *COST* here to maintain consistency with the most part of the DCOP literature, and to avoid confusion with the *VALUE* messages.

context of the ancestors being already assigned their optimal values. Eventually, the process reaches the leaves, and at this point, all agents are assigned their values from the optimal solution.

Remark 4 *The problem of rediscovering the solution is a problem that apparently occurs in all* distributed *search algorithms that do not do full caching. However, this does not occur in a centralized algorithm, as in that case "the best solution so far" can be stored, and retrieved at the end of the process. This is another complication that has to be solved in order to have very efficient* distributed *search algorithms.*

Proposition 4 (dAO-opt complexity) *dAO-opt (Algorithm 1) requires a number of messages which is exponential in the depth of the DFS tree used. Message size and memory requirements are linear for each agent.*

PROOF. Straightforward from the centralized case, as dAO-opt simulates a synchronized AND/OR search in a distributed fashion[54]. □

It becomes apparent that it is desirable to find DFS arrangements with low depth, as the worst case complexity of dAO-opt depends on this parameter. We review in Section 3.4.2.1 some existing heuristics for generating shallow pseudotrees.

3.1.1.3 dAOBB: AND/OR Branch and Bound for DCOP

The simple *dAO-opt* does not take advantage of any pruning techniques, and therefore it explores the full search space. This is not a problem for enumeration tasks such as counting solutions or computing the probability of evidence[135]. However, for simpler tasks like finding the optimal solution, traversing the whole search space is not always required, and implies spending unnecessary effort. Marinescu and Dechter introduced in[131] an adaptation of the classical branch and bound algorithm on a pseudotree, thus yielding an algorithm called AOBB (AND/OR Branch and Bound). AOBB was shown in[131] to be quite efficient in a centralized setting, especially when using minibucket heuristics for generating tighter upper bounds.

We present here *dAOBB*, an adaptation of the AOBB algorithm for the distributed case. The algorithm is described in Algorithm 2. As in *dAO-opt*, the search starts top-down, with agents assigning themselves values, and sending *EVAL* messages to their children. However, in order to be able to prune parts of the search space according to the branch and bound scheme, each agent X_i needs some information about the current cost structure:

1. the cost $cpa(X_i, \langle Sep_i \rangle)$ already accumulated by the current partial assignment from the root, to the current agent.

2. the cost $local\_cost(v_i^j, \langle Sep_i \rangle)$ of each one of the values of X_i, given the current values of X_i's ancestors

3. the cost of the best currently known solution of the subtree rooted at X_i, i.e. the current upper bound

Definition 9 (Cost of current Partial Assignment - CPA) *We define the cost of the current partial assignment $cpa(\langle Sep_i \rangle)$ as the cost accumulated from all the cost functions along the current branch which are fully instantiated:*

$$cpa(\langle Sep_i \rangle) = \sum_{X_j ancestor of X_i} local\_cost(v_j, \langle Sep_j \rangle) \tag{3.1}$$

The CPA represents the sum of the cost functions encountered from the root to the parent of X_i, which are fully instantiated. Normally (e.g. in *dAO-opt*, or ADOPT), agents do not have access to these costs incurred above themselves. Therefore, we introduce a modification to the *EVAL* messages: now, they also include the cost of the partial assignment so far. These partial costs accumulate and propagate down together with the *EVAL* messages sent from agents to their children.

The CPA received from the parent in the *EVAL* message, plus the evaluation $local\_cost(v_i^j)$, give the cost of the current partial assignment, extended by $X_i = v_i^j$: $cost(\langle Sep_i, X_i = v_i^j \rangle = cpa(X_i, \langle Sep_i \rangle) + local\_cost(v_i^j, \langle Sep_i, X_i = v_i^j \rangle)$. Clearly, this cost is a lower bound on the cost of any complete assignment, for any instantiation of the variables in the subtree of X_i.

The propagation of the *EVAL* messages proceeds down the DFS tree, towards the leaves as in *dAO-opt*. Initially, all agents start with lower bounds equal to the cost of the current partial assignment (see Algorithm 2, line 7), and infinite upper bounds (line 4).

When a leaf receives an *EVAL* message, it computes the cost of each one of its values with the constraints it has with ancestors, just like a normal agent. As the leaf has no children, it can simply select the best value from its domain (lowest cost with ancestors), and reply back to the parent with a *COST* message that reports this lowest cost.

When an agent X_i receives COST messages from its children, it does the following:

1. sum up all COST messages from children - lines 9-12. The result is the optimal cost for all the subtree rooted at X_i, *for the current instantiation of* Sep_i.

2. If this optimal cost improves the current upper bound, then update the upper bound as a better solution has been found - line 13.

3. Consider next untried value $v_i^j \in dom(X_i)$. Compute its lower bound: $LB(v_i^j) = cpa(X_i, \langle Sep_i \rangle) + local\_cost(v_i^j)$. If $LB(v_i^j) > UB$ (i.e. the minimal cost incurred by choosing $X_i = v_i^j$ is larger

than the best solution found so far), then it is useless to try assigning $X_i = v_i^j$, as this could not lead to a better solution. Therefore, prune $X_i = v_i^j$, and try another value.

4. Otherwise, try $X_i = v_i^j$. Send $EVAL(X_i = v_i^j, LB(v_i^j))$ to all children ($LB(v_i^j)$, computed as $LB(v_i^j) = cost + cost(v_i^j)$ represents the cost of the current partial assignment extended by $X_i = v_i^j$). Wait for *COST* replies, and repeat previous step until no more values to try.

5. when all values are tried, pick optimal value v_i^* that minimizes $total\_cost(v_i^*) = cost + cost(v_i^*) + \sum_{C_i} COST_{C_i}(v_i^*)$. Send to parent P_i a cost message: $COST_{X_i}(total\_cost(v_i^*))$

6. when parent sends another *EVAL* message, reset bounds, and repeat the process (cycle through all the values in own domain).

When the root has received *COST* messages for all its values (or pruned them), the optimal cost for all the problem has been found.

Remark 5 *As with dAO-Opt (Section 3.1.1.2), when caching is not allowed, one needs to revisit parts of the search space to rediscover the optimal solutions for certain subtrees. However, in the case of dAOBB the problem may not be as severe as for dAO-Opt, as the pruning mechanism may limit the amount of extra effort required.*

Proposition 5 (dAOBB complexity) *dAOBB (Algorithm 2) requires a number of messages which is exponential in the depth of the DFS tree used. Message size and memory requirements are linear for each agent.*

PROOF. Follows from Proposition 4, and from the fact that the branch and bound scheme has the same worst case complexity as the AND/OR search. □

dAOBB with heuristics: It is well known that good initial bounds are essential to the efficiency of a branch and bound scheme. For this purpose, the centralized AOBB algorithm has been enhanced in[131] with both static and dynamic heuristics based on mini-buckets. The static bounds based on minibuckets are computed by running a bounded inference phase in a preprocessing step, and saving the bounds obtained as lower bounds, which are then used in the main branch and bound phase. The dynamic bounds are computed by interleaving the bounded inference phase with the branch and bound process, and continuously updating the lower bounds. Petcu and Faltings[158] introduce A-DPOP, an adaptation of the minibucket scheme to a distributed setting; A-DPOP can be easily used in conjunction with dAOBB to produce better bounds, either static or dynamic.

dAOBB(i): Distributed AND/OR Branch and Bound with caching Similarly to AOBB with caching[132], one can extend dAOBB to equip it with a customizable and adaptable caching scheme.

Algorithm 2 *dAOBB - distributed AO B&B search for cost minimization.*

dAOBB($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): each agent X_i does:

Construct DFS tree; after completion, X_i knows P_i, PP_i, C_i, PC_i
1 **if** X_i *is root* **then** do EVAL
2 **else** wait for EVAL messages from parent

procedure EVAL: X_i received an EVAL($\langle Sep_i \rangle, cost$) message from P_i
3 let $\langle Sep_i \rangle$ be the received assignment of variables in Sep_i
4 $UB \leftarrow \infty$
5 **forall** $v_i \in dom(X_i)$ **do**
6 $cost(v_i) \leftarrow$ cost between X_i and its ancestors, when $X_i \leftarrow v_i$ and $Sep_i \leftarrow \langle Sep_i \rangle$
7 $LB(v_i) = cost + cost(v_i)$
8 **if** $LB(v_i) < UB$ **then**
9 **forall** $X_j \in C_i$ **do**
10 send EVAL($\langle Sep_{X_j} \rangle, LB(v_i)$) message to X_j containing the current $\langle Sep_{X_j} \rangle$
11 wait for the $COST_{X_j}(v_i)$ replies from children
12 $cost(v_i) += COST_{X_j}(v_i)$
13 **if** $cost(v_i) < UB$ **then** $UB = cost(v_i)$
14 pick v_i^* s.t. $v_i^* = argmin_{v_i \in dom(X_i)}(cost(v_i))$
15 **if** X_i *is root* **then** v_i^* is the root's value in the optimal solution
16 **else** send $COST(cost(v_i^*))$ to P_i

The user can specify the parameter i which represents the maximal size of any cache table; subsequently, each agent X_i caches in its table results of searches for a subset of variables in its Sep_i which is bounded by i. Previous search results can be retrieved from the cache; however, whenever one of the agents in Sep_i not included in the cache changes its value, the cache table has to be purged and recomputed. Depending on the structure of the problem, *dAOBB(i)* can provide exponential speedups over simple *dAOBB*.

Concretely, the caching mechanism can be added to dAOBB by making the following changes to Algorithm 2:

- initialize cache of size d^i after line 2;
- in *EVAL* (after line 3) purge cache if any agent in Sep_i which is not in the cache changed its value in $\langle Sep_i \rangle$;
- in *EVAL* (after line 3) check if the received assignment for $\langle Sep_i \rangle$ is found in the cache; if so, return it with its corresponding cost. Otherwise, after line 14 cache $(\langle Sep_i, v_i^* \rangle, cost(v_i))$

Proposition 6 (dAOBB(i) complexity) *dAOBB(i) requires at most $O(exp(i))$ memory at each agent. Messages are of linear size. The number of messages required varies with the level of caching:*

$O(exp(w))$ when using full caching (i.e. $i \geq w$) and $O(exp(depth))$ when using bounded caching (i.e. $i < w$).

PROOF. Follows straightforwardly from the centralized case[132]. □

Similar to dAO-opt, $dAOBB(i)$ can also benefit from DFS arrangements with low depth (see Section 3.4.2.1 for some heuristics). However, considering that the number of messages depends also on the induced width (when full caching is used) it becomes apparent that it is desirable to minimize not only the DFS depth, but also the induced width.

3.1.2 Asynchronous search algorithms

The vast majority of algorithms developed so far for DisCSP/DCOP are asynchronous algorithms. Asynchrony is appealing for distributed algorithms for a number of reasons. First, asynchrony can offer in principle a better distribution of the computation between the agents involved (all agents can execute in parallel, and do not necessarily have to wait for messages from their peers). Second, asynchronous algorithms are in principle less sensitive to message delays and message loss, as agents execute even without necessarily having received the most up to date messages from their peers.

We start this section with a short review of asynchronous algorithms for distributed constraint *satisfaction*. Next, we move to algorithms for distributed constraint *optimization* and describe ADOPT, NCBB and AFB.

3.1.2.1 Asynchronous search for DisCSP

This section describes existing asynchronous approaches for Distributed Constraint *Satisfaction* Problems. This is by far the area which has received the most attention since the beginnings of the distributed constrain reasoning field, in the early nineties. Undoubtedly, the most influential piece of work is Yokoo's Asynchronous Backtracking (ABT) algorithm, which represented the basis for many subsequent developments. We describe this algorithm in the following.

Asynchronous Backtracking (ABT) Asynchronous Backtracking (ABT[224]) is the first asynchronous algorithm that has been proposed for DisCSP. ABT laid the foundations of DisCSP, being the first algorithm to allow agents to execute concurrently and asynchronously.

In ABT, agents are ordered linearly. They assign values to their variables concurrently and asynchronously, and announce the assignments to their lower-priority neighbors via `ok?` messages. When an agent receives an `ok?` message, it updates its *agent view* [5] and tries to find a compatible value for

[5] A data structure holding the agent's view of the current assignment of agents of higher priority

its variable. If it can, it announces this to lower priority agents with an `ok?` message, otherwise, it backtracks with a `nogood` message. When receiving a `nogood` message, an agent tries to find another value for its value, compatible with its own agent view. If it cannot, it backtracks with a `nogood`, and so on. The algorithm terminates if an empty nogood is discovered (the problem has no solution), or if quiescence is reached, in case a solution is found. Note that detecting that a solution was found requires an additional termination detection algorithm, which may introduce some overhead.

ABT is sound and complete, and its complexity is polynomial amount of memory, and exponential number of messages in the worst case. ABT has been extensively studied since its original publication by Yokoo in '92[224], and much of the later work in DisCSP is based on it.

Asynchronous Weak Commitment (AWC): Asynchronous Weak Commitment (AWC[223]) is an alternative to ABT which was proposed in order to simulate the dynamic variable ordering heuristics from the centralized case, which have been shown to offer important performance improvements in some cases. Specifically, whenever an agent initiates a backtrack, it takes the first position in the ordering. This step is designed to refocus the algorithm on the newly discovered difficult part of the search space. AWC is shown to be more efficient than ABT on difficult problems[86, 223]. However, in this case, AWC must store all nogoods discovered during search to guarantee completeness, which makes it space-exponential in the size of the problem in the worst case. On a side note, Yokoo and Hirayama[227] introduce a modification of AWC which deals with complex local problems, i.e. an agent owns several variables as opposed to a single one.

Dynamic Variable Reordering: In order to allow distributed search to benefit from dynamic variable ordering heuristics like AWC, but without AWC's exponential space problems, variable reordering techniques have been developed for the ABT algorithm in[193, 199, 202], and then also in[242]. These techniques work by allowing only for a limited type of reordering, namely each agent can impose new orderings for agents below itself in the ordering, and inform these lower priority agents of the new ordering. Upon being announced of a change in the ordering, a lower priority agent updates its `agent_view`, and discards obsolete nogoods. A more advanced reordering protocol is introduced by Silaghi et al.[201]. This protocol allows for general reorderings, thus being able to simulate AWC with polynomial space requirements.

Asynchronous Aggregations in Search (AAS): AAS (asynchronous aggregations in search) [197] is an algorithm that operates on the dual model of the CSP, i.e. where agents own and control the constraints, not the variables, which are public. The domains are now tuples of assignments of variables from the original CSP formulation, and can be large. AAS exploits the fact that in cases where variables have large domains, it could happen that several values in the domain behave similarly with respect to constraints. Thus, it can be beneficial to group several values in equivalent sets, and perform ABT on

such a modified problem, and managing the grouping into sets dynamically during search.

Asynchronous Consistency Techniques: Consistency techniques have been shown to be very effective in centralized CSP, and have been also implemented in distributed settings[26, 95, 138, 139, 195, 200]. Asynchronous Forward-Checking introduced by Meisels and Zivan[139] works by having agents perform backtracking sequentially, and announcing their assignments in parallel to all other agents lower in the ordering, which perform forward checking in parallel. Hamadi proposes a distributed arc consistency algorithm in[95]. Silaghi introduces MHDC[195], an algorithm which maintains arc consistency during search in AAS, which is shown to improve AAS's performance significantly.

Concurrent Search: Multiple search processes operating concurrently and exchanging information have also been investigated. The idea is to launch parallel search processes that explore different parts of the search space, and let them communicate relevant nogoods between themselves, such that they avoid exploring the same dead ends. [176, 241] report promising results.

The vast majority of algorithms that do not operate on a DFS require communication between non-neighbors. This is also the case of ABT (requested via *add-link* messages) and derivatives, AWC, DisFC, ConcBT, etc. All these algorithms violate our assumption from Section 2.2. An extension to ABT proposed by Bessière and colleagues[22] eliminates the need to add links, but incurs a performance hit for doing so.

Distributed Local Search: A local search method called the Distributed Breakout Algorithm[226] has also been developed. DBA is not complete, and works only for satisfaction problems, but in some cases it can find solutions very fast, and it also exhibits anytime behaviour for overconstrained problems. In DBA, agents execute a hillclimbing algorithm in parallel, and try to escape from *quasi-local minima* [6] using the breakout method[143]. In DBA, agents initially choose arbitrary values for their variables, and announce their choices to their neighbors with `ok?` messages. Subsequently, when receiving `ok?` messages, each agent evaluates the number of conflicts its current assignment produces with the assignments of its neighbors. The agent (internally) evaluates what reduction in the number of conflicts it could make by changing its value, and advertises this possible improvement to its neighbors with an `improve` message. Neighboring agents thus exchange `improve` messages, and the one with the highest improvement wins and actually changes its value (ties are broken according to agent ID). The cycle then repeats, with `ok?` and `improve` messages. In case a solution is found, the algorithm reaches quiescence (a termination detection is provided). If a solution is not found, or none exists, DBA cycles forever.

As with ABT, DBA has been the object of many subsequent improvements[12, 155, 157, 236, 238].

[6] In[143], the breakout method is used to escape from *global* local minima, but in a distributed setting it is difficult for the agents as a group to realize they are stuck in a global minima; thus quasi-local-minima is used as a loose, cheaper alternative.

An improvement to DBA appears in[155] which uses *interchangeabilities*[77] to try to contain conflicts, and keep them localized. This works by using neighborhood interchangeability and neighborhood partial interchangeability to select new values for the variables that already are in conflict with other variables, such that we do not risk creating new conflicts by switching to the new values. Experimental results show that the new algorithms are able to solve more problems, and with less effort, especially for difficult problems, close to the phase transition. Another improvement of DBA consisting in a value-ordering heuristic appears in[157]. This heuristic is developed in the context of resource allocation problems (e.g. sensor networks), and it works by trying to allocate the least contended resources first. This tends to produce good allocations from the beginning of the execution of DBA, and thus requires less subsequent effort. Another extension to DBA that identifies hard subproblems and solves them with a complete search algorithm, *thus guaranteeing completeness* has been proposed in[62].

Alternatively, distributed stochastic search algorithms have been proposed[6, 75, 235].

In the next sections we focus on algorithms that were specifically designed for DCOP.

3.1.2.2 ADOPT

ADOPT by Modi et al. (141]) is a backtracking based bound propagation mechanism. ADOPT was the first decentralized algorithm to guarantee optimality, while at the same time allowing the agents to operate asynchronously.

The algorithm works as follows: first, the DFS structure is created. Then, backtrack search is performed top-down, using the following value ordering heuristic: at each point in time, each agent chooses the value with the smallest lower bound. It announces its descendants of its choice via *VALUE* messages, and waits for *COST* messages to come back from the children (please refer to Figure 3.1 for a diagram that shows the message flow in ADOPT). Each agent adds the costs received from its children to the lower bound of the current value taken by the agent. If there is another value in the domain that has a smaller lower bound, the agent switches its value, and the process repeats, refining the lower bounds iteratively.

One of the innovative ideas behind ADOPT is that it achieves asynchrony by allowing the agents to change their variable values whenever they detect the *possibility* that some other values are better than the current ones (i.e. they have smaller *lower bounds*). Notice that this does not mean that the new values are guaranteed to be better. This strategy allows for asynchronous operation since the agents do not have to wait for achieving global information about *upper bounds* on cost to take their local decisions, as would normally happen in classical branch and bound.

However, abandoning partial solutions before proving their suboptimality makes it sometimes necessary to revisit several times some of the previously explored partial solutions. One solution to this problem would be to store all these partial results, and retrieve them later on, without any more search

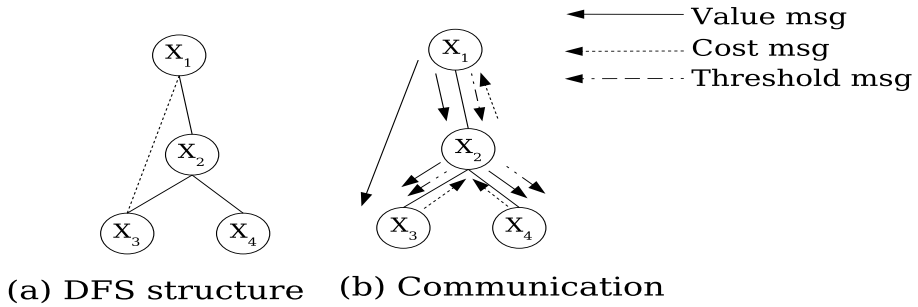


Figure 3.1: *ADOPT: (a) a simple problem graph, arranged as a DFS tree; (b) diagram showing the flow of messages: VALUE assignments are sent by agents to all their descendants in the tree; children respond to their parents with COST messages; parents send their children THRESHOLD messages.*

effort. The drawback of this approach is that the amount of memory required to do so is exponential in the width of the DFS ordering chosen. ADOPT tries to mitigate this problem by using a *backtrack threshold* which is an allowance on solution cost intended to reduce the need for backtracking, while maintaining a low memory profile (polynomial).

3.1.2.3 Non-Commitment Branch and Bound

Chechetka and Sycara propose in[33] another DCOP algorithm that operates on a DFS: NCBB (Non-Commitment Branch and Bound). This algorithm is a variant of AOBB, with the important difference that NCBB includes a parallelization technique where an agent advertises different values of itself to different children at the same time. This parallelization technique ensures that all the subtrees of any agent are working in non-intersecting parts of the search space and we do not need to worry about the solution costs between them.

Similar to dAOBB with i-bounded caching (Section 3.1.1.3), NCBB was also extended with a caching mechanism in[32].

3.1.2.4 Asynchronous Forward Bounding (AFB)

AFB[81] is also based on branch and bound, and works on a linear ordering of the variables. AFB is similar to SynchBB: agents assign their variables and generate a partial solution sequentially and synchronously. As in classic B&B, agents extend a partial solution as long as the lower bound on its cost does not exceed the global bound, which is the cost of the best solution found so far. The current partial assignment is propagated together with the cost of the best solution found so far. Each agent which receives the CPA, extends it with its local assignment, if an assignment with a lower bound smaller than the current global upper bound can be found. Otherwise, it backtracks by sending the CPA

to a former agent to revise its assignment. An agent that succeeds to extend the assignment on the CPA sends forward copies of the updated CPA, requesting all unassigned agents to compute lower bound estimations on the cost of the partial assignment. The assigning agent will receive these estimations asynchronously over time and use them to update the lower bound of the CPA. Gathering updated lower bounds from future assigning agents, may enable an agent to discover that the lower bound of the CPA it sent forward is higher than the current upper bound (i.e. inconsistent). This discovery triggers the creation of a new CPA which is a copy of the CP A it sent forward. The agent resumes the search by trying to replace its inconsistent assignment. The authors provide an experimental evaluation of AFB against SynchBB, and show that it performs better.

3.1.3 Summary of distributed search methods

The advantage of the search algorithms we have presented is that they require polynomial memory. Their downside is that they may produce a very large number of small messages, resulting in large communication overheads. As far as ADOPT is concerned, several extensions have been proposed (e.g.[127, 194]) to deal with this problem. In some cases they show improved performance over the basic ADOPT, but in the worst case, they all produce an exponential number of small messages.

If more memory is available, search can be executed more efficiently by using caching schemes like dAOBB(i) or NCBB(i); however, in the worst case search algorithms may still require $exp(w)$ messages.

3.2 Dynamic Programming (inference) in COP

Dynamic programming[15, 16] (inference) has been long recognized as a powerful paradigm for solving combinatorial optimization problems[19]. Loosely, dynamic programming works by eliminating variables one by one while computing the effect of each eliminated variable on the rest of the problem.

Bucket elimination (BE) is a unifying algorithmic framework for dynamic programming algorithms, introduced by Dechter in[50, 51]. It is applicable to any graphical model such as probabilistic and deterministic networks. The input to a BE algorithm consists of a collection of functions or relations of a reasoning problem. Given a variable ordering, the algorithm partitions the functions into buckets, each associated with a single variable. A function is placed in the bucket of its latest argument in the ordering.

The algorithm processes each bucket, top-down from the last variable to the first by a variable elimination procedure. This procedure computes a new function using combination (join) and marginalization (project, or eliminate) operators in each bucket. The new function is placed in the closest lower bucket whose variable appear in the function's scope. When the solution of the problem requires a

complete assignment (e.g., finding the most probable explanation in belief networks) a second, bottom-up phase, assigns a value to each variable along the ordering, consulting the functions created during the top-down phase.

3.2.1 BTE

BTE (bucket tree elimination) is a centralized algorithm introduced by Kask et al. ([107]) and Shenoy ([190]). This algorithm leverages the basic bucket elimination mechanism[50] by operating on a *bucket tree*, and performing bucket elimination on this tree in both top-down and bottom-up phases.

This requires twice the amount of effort as the normal bucket elimination scheme, but the advantage is that it enables complex tasks like belief updating in a Bayesian network, or computing optimal utilities for each value of each variable in the problem. In these cases, the normal bucket elimination scheme would have to be applied once for each variable in the problem, thus increasing the complexity of the process linearly with the number of variables.

3.3 Partial Centralization: Optimal Asynchronous Partial Overlay (OptAPO)

Optimal Asynchronous Partial Overlay (OptAPO[129]) is a sound and optimal algorithm for solving DCOPs that uses dynamic, partial centralization (DPC). Conceptually, DPC is a technique that discovers difficult portions of a shared problem through trial and error and centralizes these sub-problems into a mediating agent in order to take advantage of a fast, centralized solver. Overall, the protocol exhibits an early, very parallel hill climbing behavior which progressively transitions into a more deliberate, controlled search for an optimal solution. In the limit, depending on the difficulty of the problem and the tightness of the interdependence between the variables, one or more agents may end up centralizing the entire problem in order to guarantee that an optimal solution has been found.

The authors report that OptAPO's message complexity is significantly smaller than ADOPT's[129]. However, it is possible that several mediators solve overlapping problems, thus needlessly duplicating effort. This has been shown in[169] to cause scalability problems for OptAPO, especially on dense problems. Furthermore, the asynchronous and dynamic nature of the mediation sessions make it impossible to predict what will be centralized where, how much of the problem will be eventually centralized, or how big a computational burden the mediators have to carry. It has been reported by Davin and Modi in[44] that often a handful of nodes centralize most of the problem, and therefore carry out most of the computation.

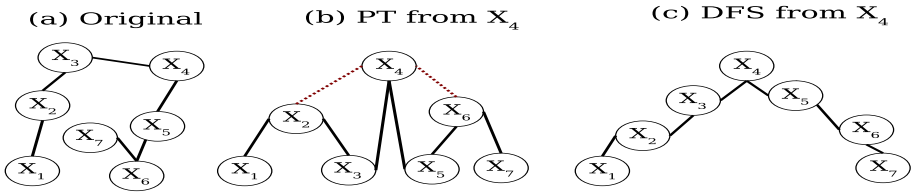


Figure 3.2: *A simple problem (a), a possible pseudotree(b), and a rooted DFS tree(c). Notice that (c)* ***is*** *a pseudotree, while (b)* ***is not*** *a DFS tree.*

3.4 Pseudotrees / Depth-First Search Trees

Definition 10 (Pseudo-tree) *A pseudo-tree arrangement of a graph G is a rooted tree with the same nodes as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g.* X_0 *and* X_{11} *in Figure 3.3).*

Notice that Definition 10 allows for the pseudotree to be a rooted tree with *more edges than the original graph G*. For example, consider a problem that is a chain with 7 nodes: $X_1 \dots X_7$ (see Figure 3.2(a)). A pseudotree for this problem can be as in Figure 3.2(b) or (c). Notice that the pseudotree in Figure 3.2(b) requires the addition of the two dotted edges $X_4 - X_2$ and $X_4 - X_6$, while the one in Figure 3.2(c) contains only edges from the original graph.

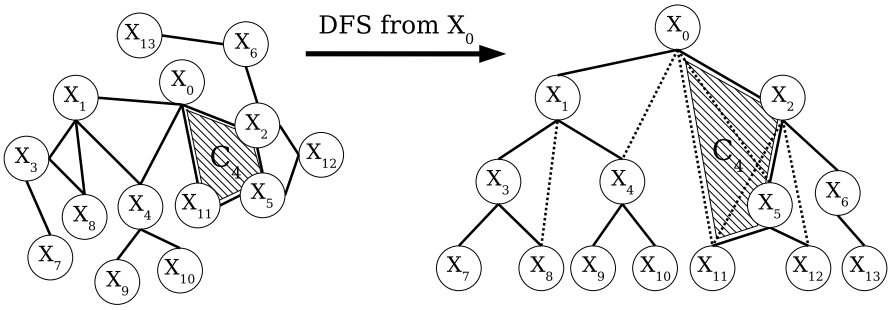
The use of pseudotrees in constraint satisfaction was first introduced by Freuder in[78], and subsequently exploited in **(**13, 38, 39, 54, 141]). The idea is that nodes lying in different branches of the DFS tree become conditionally independent when all their ancestors are removed. It is thus possible to perform search in parallel on these independent branches. Specifically, one starts instantiating nodes top-down (starting from the root); then for each node, once it is instantiated, its subtrees become completely independent, and can be explored in parallel.

3.4.1 DFS trees

A special case of a pseudotree is when all arcs of the pseudotree belong to the original graph. It is easy to see that this special class can be generated by a depth-first search traversal of the graph. Therefore, these are called DFS trees. Formally,

Definition 11 (DFS tree) *A DFS arrangement of a graph G is a rooted tree with the same nodes* ***and edges*** *as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g.* X_0 *and* X_{11} *in Figure 3.3).*

It is well known that a depth-first traversal of a graph produces a pseudotree arrangement; DFS trees are thus a subclass of pseudotrees. However, there are pseudotree arrangements that are not DFS



(a) a simple graph (b) a depth-first traversal from X_0

Figure 3.3: *A problem graph and a rooted DFS tree. Non-binary constraints like C_4 are treated as cliques of the variables involved. Tree edges are solid lines, while back-edges are dashed lines.*

trees, for example the one from Figure 3.2(b). For the purposes of distributed optimization algorithms, we will focus on DFS structures, because we assume that only neighboring agents can communicate directly [7] (see Section 2.2.2). In addition, it is well understood how to generate a DFS tree distributedly, while it is far less clear for pseudotrees that are not DFS trees. Nevertheless, all the algorithms we will present can, in principle, work on general pseudotree structures once we relax this communication assumption.

Figure 3.3(b) shows an example of a DFS tree for the graph in Figure 3.3(a) that we shall refer to in the rest of this section (ignore for now the shaded areas). We distinguish between *tree edges*, shown as solid lines (e.g. $X_8 - X_3$), and *back edges*, shown as dashed lines (e.g. $X_8 - X_1$, $X_{12} - X_2$). We call a path in the graph that is entirely made of tree edges, a *tree-path*. A tree-path that connects a node with one of its descendants is called a *branch*. A *tree-path associated with a back-edge* is the tree-path connecting the two nodes connected by the back-edge.

Definition 12 (DFS concepts) *Given a rooted DFS tree T of a graph G, for each node X_i in the tree, we define:*

- *The* **children C_i / parent P_i** *of node X_i: these are the descendants / ancestor of X_i which are connected to X_i through a tree edge (e.g. $P_4 = X_1$, $C_1 = \{X_3, X_4\}$).*
- *The* **pseudo-parents PP_i** *of node X_i are X_i's ancestors that are connected to X_i through back-edges ($PP_8 = \{X_1\}$). Notice that $P_i \notin PP_i$.*
- *The* **pseudo-children PC_i** *of node X_i are X_i's descendants directly connected to X_i through back-edges (e.g. $PC_0 = \{X_4, X_5, X_{11}\}$).*

[7] In the example problem from Figure 3.2, if one uses the pseudotree arrangement from Figure 3.2(b), the 2 pairs of agents $X_4 - X_2$ and $X_4 - X_6$ would be required to communicate even though they are not neighbors in the interaction graph.

- Sep_i *is the* **separator** *of node* X_i*: all ancestors of* X_i *which are connected with* X_i *or with descendants of* X_i *(e.g.* $Sep_3 = \{X_1\}$*,* $Sep_5 = \{X_0, X_2\}$ *and* $Sep_8 = \{X_1, X_3\}$*); otherwise stated, given a DFS tree,* Sep_i *is the minimal set of ancestors of* X_i *whose removal completely disconnects the subtree rooted at* X_i *from the rest of the problem. For trees,* $Sep_i = \{P_i\}, \forall X_i \in \mathcal{X}$*.*

Each node X_i can easily determine its separator Sep_i as the union of: (a) separators received from its children, and (b) its parent and pseudoparents, minus itself (see Definition 12). Formally,

$$Sep_i = \cup_{X_j \in C_i} Sep_j \cup P_i \cup PP_i \setminus X_i. \tag{3.2}$$

Given a DFS arrangement of a constraint graph, we define the *depth* of the DFS tree as the number of nodes on the longest branch. Additionally, the *induced width*[51, 110, 111] of a graph G given an ordering $o = X_1, \ldots, X_n$ is defined as follows:

Definition 13 (Induced Width) *Given a graph* G *and an ordering* $o = X_1, \ldots, X_n$ *on its nodes, the* induced width *of the graph according to this ordering is defined as follows: we process all nodes in the reverse order of* o*. When processing a node, we connect all its neighbors which precede it in the ordering* o*. The width of the current node is given by the number of its induced neighbors which precede it in the ordering* o*. The* induced width of the ordering o *is the largest width of any node in ordering* o*.*

When considering as an ordering o the depth-first traversal of the nodes in G along a given DFS arrangement of G, we have:

Proposition 7 *The induced width of a graph* G *along a given DFS arrangement is equal to the size of the largest separator of any node in the DFS arrangement.*

PROOF. Consider Definition 13 of the width of each node in the DFS arrangement. We process the nodes in G in the reverse DFS order. When processing a node as in Definition 13, we connect all its neighbors in G which are its ancestors in the DFS, i.e. we connect its parent with all its pseudoparents. We do this recursively in reverse DFS order, from the leaves until we reach the root. At the end of the process, for each node X in the DFS, we will have an (induced) neighboring relation between X and all its ancestors which are connected in G with either X or any of its children. This means that using Definition 13 for the width of a node, we fall exactly on the Definition 12 of the separator of the node. Therefore, the induced width of the DFS ordering equals the size of the largest separator of any node in the DFS, as in Definition 12. □

3.4.1.1 Distributed DFS generation: a simple algorithm

Generating DFS trees in a distributed manner is a task that has received a lot of attention, and there are many algorithms available: for example Collin and Dolev[40], Barbosa[10], Cidon[36], Cheung[35] to name just a few. For completeness, we specify a possible distributed DFS algorithm, which is similar to Cheung[35]. We present this simple algorithm in Algorithm 3, and we will assume it is executed in a preprocessing phase by all the algorithms that we will present for static optimization. When we move to dynamic problems in Chapter 9, we will assume the self-stabilizing algorithm of Collin and Dolev[40]. In Section 3.4.2, we will extend Algorithm 3 with different heuristics that produce better quality DFS trees.

Algorithm 3 starts with each agent X_i identifying its set of neighbors, $Ngh(X_i)$, as all other agents X_j with whom X_i shares a relation or a constraint (see Chapter 2). Each agent then labels internally its neighbors as *not-visited*. One of the agents in the graph is designated as the *root*, using for example a leader election algorithm like[2], or simply picking the agent with the lowest/highest ID.

The root then initiates the propagation of a *token*, which is a unique message that will be circulated to all the agents in the graph, thus "visiting" them. Initially, the token contains just the ID of the root. The root sends it to one of its neighbors, and waits for its return before sending it to each one of its (still) unvisited neighbors. When an agent X_i first receives the token, it marks the sender as its *parent*. All neighbors of X_i contained in the token are marked as X_i's pseudoparents (PP_i).

After this, X_i adds its own ID to the token, and sends the token *in turn* to each one of its *not-visited* neighbors X_j, which become its *children*. Every time an agent receives the token from one of its neighbors, it marks the sender as *visited*. The token can return either from X_j (the child to whom X_i has sent it in the first place), or from another neighbor, X_k. In the latter case, it means that there is a cycle in the subtree, and X_k is marked as a *pseudochild*.

When all its neighbors are marked *visited*, X_i has finished exploring all its subtree. X_i then removes its own ID from the token, and sends the token back to its parent; the process is finished for X_i. When the root has marked all its neighbors *visited*, the entire DFS construction process is over.

Proposition 8 *Algorithm 3 produces a correct DFS arrangement which is maintained in a distributed fashion.*

PROOF. Algorithm 3 is correct because it simulates exactly a centralized depth-first search process. Furthermore, due to the fact that each node adds its ID to the token when sending it to its children, and then removes it when sending it back to its parent, the structure of the whole problem remains hidden from individual agents. Each agent only knows its position in the tree, which is given by its knowledge of its parent, children, pseudoparents, and pseudochildren. □

Proposition 9 *Algorithm 3 produces $2 \times |E|$ messages of linear size, where $|E|$ is the number of edges*

in the graph.

PROOF. It is easy to see that there is exactly 1 *DFS* message going in each direction through each edge: once when the parent node sends the token to the child the first time, and one more time when the child has finished exploring its subtree and returns the token. Thus the total number of messages is $2 \times |E|$. The size of these messages is linear, the largest one having a number of IDs in the context that equals the height of the DFS tree. □

Remark 6 (Non-binary constraints) *Non-binary constraints are automatically handled correctly by Algorithm 3 as a result of the fact that all agents involved in a constraint or relation (be it binary or non-binary) label each other as neighbors (Chapter 2). Then, in Algorithm 3, the* `for` *loop in line 6 ensures that the first agent involved in a non-binary constraint, when receiving the token, will subsequently pass it to all other agents involved in that constraint, thus making them its descendants. This ensures that there are no cross-edges between different subtrees and the DFS is correctly constructed. For example, in Figure 3.3 (left), there is a 4-ary constraint C_4 involving $\{X_0, X_2, X_5, X_{11}\}$. By Definition 3, this implies that $\{X_0, X_2, X_5, X_{11}\}$ are neighbors, and in the DFS construction process and they will appear along the same branch in the tree. This produces the result in Figure 3.3 (right).*

For the rest of this chapter, we will assume that all the algorithms presented will use Algorithm 3 in a preprocessing phase, to establish the required DFS structure.

Example 5 (Execution of DFS construction Algorithm 3) *Please refer to Figure 3.3 for an example. Without loss of generality, let us assume that agent X_0 has been chosen as the root of the DFS tree. X_0 sends a token with just its ID, DFS[0], to one of its neighbors (e.g. to X_1). X_0 marks X_1 as its child, and X_1 marks X_0 as its parent ($P(X_1) = X_0$). X_1 adds its own id to the context of the received DFS message, and then sends it to an unvisited neighbor (e.g. to X_4).*

X_4 receives DFS[0,1] from X_1 and marks it as its parent. Now, since X_0 is X_4's neighbor, and X_0 is also present in the context of the message that X_4 received from X_1, X_4 marks X_0 as its pseudoparent, and sends the message DFS[0,1,4] to X_0. Thus, X_0 can also mark X_4 as its pseudochild.

X_4 continues by sending DFS[0,1,4] to X_9, receiving it back, and to X_{10} and receiving it back. At this point, X_4 has finished exploring its subtree (all neighbors are visited), so it sends back to X_1 a DFS[0,1] message, which informs X_1 that the discovery of the subtree hanging from X_4 is finished. X_1 can then continue with the exploration of its other subtree, and sends its DFS[0,1] message to X_3. X_3 sends DFS[0,1,3] to X_8, which marks X_1 as its pseudoparent and sends it DFS[0,1,3,8], which means that X_1 can also mark X_8 as its pseudochild, and so on.

Algorithm 3 *A DFS construction algorithm for DCOP.*

Inputs: each agent X_i knows all its neighbors $X_j \in Ngh(X_i)$
Outputs: each X_i labels all its neighbors as either P_i, PP_i, C_i, PC_i.

Procedure Initialization
1 The agents $\mathcal{X}$ choose one of them, X_0, as the root (e.g. via leader election).
2 All agents execute procedure **Token_Passing**

Procedure Token Passing (performed by each "virtual agent" X_i)

if X_i *is root* **then** $P_i = null$; create empty token $DFS = \emptyset$
3 **else** DFS=**Handle_incoming_tokens()**
4 let $DFS_i = DFS \cup \{X_i\}$
5 *[Optional: sort* $Ngh(X_i)$ *according to heuristic (see Section 3.4.2)]*
6 **forall** $X_l \in Ngh(X_i)$ **do**
  if X_l *not visited yet* **then**
7 add X_l to C_i
8 send DFS_i to X_l
9 wait for DFS_l to return from X_l
10 X_i's subtree completely explored; remove X_i from DFS_i and send it back to P_i

Procedure Handle_incoming_tokens()

11 wait for any incoming DFS_l message; let X_l be the sender; mark X_l *visited*
if *this is the first* DFS *message (i.e.* X_l *is my parent)* **then**
12 $P_i = X_l$; $PP_i = \{X_k \neq P_i | X_k \in Ngh(X_i) \cap DFS_l\}$
else
13 *[Optional: sort unvisited neighbors according to heuristic (see Section 3.4.2)]*
  if $X_l \in C_i$ *(i.e. this is a DFS message returning from a child)* **then**
14 continue with other neighbors
15 **else** *(i.e. this is a DFS message coming from a pseudochild)*; add X_l to PC_i

3.4.2 Heuristics for finding *good* DFS trees

The complexity of all the algorithms we will present in the following sections depends on the particular DFS tree we choose. In the case of linear-size search-based algorithms (Section 3.1), the complexity is time exponential in the depth of the DFS tree. Dynamic programming methods (Section 3.2) on the other hand are time and space exponential in the width of the DFS tree. Therefore, depending on the algorithm to be used, one would like to have either the minimal depth DFS tree, or the minimal width tree. However, it has been shown that finding either of these is an NP-hard problem. Typically, one must settle for an approximation of the best DFS tree, that can be obtained using some heuristic generation process. The DFS construction Algorithm 3 can be parametrized with 2 parameters: the start agent (the root), and a heuristic function that each agent uses to decide at each step to which unvisited neighbor it

will send the token next.

There exist already a number of heuristics to generate good DFS trees in the centralized case. However, implementing these techniques in a distributed fashion may not be easy, or even feasible. We will discuss some possibilities for distributed adaptations, for search algorithms requiring shallow trees in Section 3.4.2.1 and for inference algorithms requiring trees with low width in Section 3.4.2.2.

3.4.2.1 Heuristics for generating low-depth DFS trees for search algorithms

While many algorithms exist for generating shallow DFS trees in the centralized case (e.g.[127, 132, 135]), it is unclear how to implement them in a distributed way, and little work has been done in this area. Chechetka and Sycara introduced in[31] the first distributed algorithm that constructs a pseudotree[78] using a heuristic designed to minimize the depth of the pseudotree. The algorithm works well, but in general it does not produce DFS trees, rather pseudotrees, thus violating our requirement from Section 2.2.2.

Actually, the fact that we require DFS trees as opposed to just any pseudotree means that search algorithms can be arbitrarily bad compared to dynamic programming ones. To see this, consider a simple example of a ring constraint network with n agents. Any DFS arrangement of such a network will have depth n, thus making search algorithms run in time exponential in n (runtime is $O(d^n)$). In contrast, a dynamic programming algorithm like DPOP would only be exponential in the width of the DFS, which is 2 for a ring, thus offering an exponential speedup (runtime is $O(d^2)$).

3.4.2.2 Heuristics for generating low-width DFS trees for dynamic programming

The objective of these methods is to produce the DFS arrangement with the lowest induced width. In a centralized setting, the most common heuristics for this problem are the following: the *maximum cardinality set*[207], the *maximum degree*[207], and the *min-fill* heuristic[110]. The min-fill heuristic does not produce in general pseudotree orderings (much less DFS ones), and is difficult to implement in a distributed setting because it would require coordination at each step between all the remaining agents in order to decide which one should be considered next in the elimination ordering. In the following we describe distributed adaptations of the maximum cardinality set and max-degree heuristics.

MCN: maximum connected node A heuristic called *the most connected node (MCN)* (also known as *max-degree*) has been proved quite effective. MCN was introduced by[207], and subsequently re-explored in[25, 84, 111, 127]. This heuristic works as follows: the agent with the maximum number of neighbors is selected as the root (ties are broken by picking the agent with the lowest ID). Afterwards, the process proceeds by visiting at each step neighboring agents with the highest number of neighbors (ties are again broken by picking the neighbor with the lowest ID).

Concretely, the process is implemented by changing the DFS algorithm 3 in two places. First, in step 1 each agent broadcasts the number of its neighbors; the agent ranked highest is chosen as the root. Second, step 5 is implemented by having each agent sort the list of its neighbors, the most connected ones first. The rest proceeds as normal.

MCS: maximum cardinality set adapted to DFS trees The *maximum cardinality set* heuristic was introduced by [207], and was subsequently used in many other contexts like [25, 84, 111]. This heuristic is designed to find low-width elimination orders for variable elimination procedures. It works by selecting some agent as the first one to be eliminated, and adding it to the set S of visited agents. Then, each agent not in S is considered in turn. The one that has the most number of neighbors already in S is selected to be eliminated next, and is placed in the set S. Ties are broken randomly (or by agent ID). The process is repeated until all agents are in S.

MCS as was originally described in [207] does not produce a DFS ordering of the agents in the graph. Therefore, we propose in the following a simple adaptation of the DFS generation Algorithm 3 that takes advantage of the MCS heuristic. We replace the DFS message handling code from Algorithm 3 (lines 11-15) with the following process, which is intended to simulate the MCS heuristic: Whenever an agent X_i receives a DFS message from one of its neighbors , X_i does the following:

- select its neighbors that are not either already visited, nor in the context of the DFS message : these are agents not yet visited, future children/pseudochildren;
- ask each one of them how many of their neighbors are already in the context of the DFS message;
- send the DFS token next to the neighbor which replies with the highest number;

Part II

The DPOP Algorithm

Chapter 4

DPOP: A Dynamic Programming Optimization Protocol for DCOP

"Good things come in large packages."

—

In this chapter we introduce the DPOP algorithm for DCOP. DPOP is an algorithm based on dynamic programming[19] which performs bucket elimination[49] on a DFS tree in a distributed fashion. DPOP's main advantage is that it requires only a linear number of messages, thus introducing exponentially less network overhead than search algorithms when applied in a distributed setting. Its complexity lies in the size of the UTIL messages, which is bounded exponentially by the induced width *of the DFS ordering chosen. DPOP is therefore an excellent choice for solving DCOP in case the problems have low induced width.*

In case the problems have high induced width and DPOP is unfeasible, other techniques must be explored. The whole part III of this thesis (Chapters 6, 7 and 8) discusses techniques that deal with the exponential space problem in different ways, offering different tradeoffs.

For the centralized case, we have reviewed in Section 3.2.1 the BTE algorithm introduced by Kask et al. ([107]) and Shenoy ([190]). BTE is a general algorithm which operates on any variable ordering (which is assumed to be given as input). BTE then creates a pseudotree which corresponds to this ordering, and operates on this pseudotree. The issue in a multiagent setting is that operating on arbitrary pseudotrees (i.e. non DFS) breaks the assumption that only neighbors can communicate directly (see Section 2.2).

Therefore, this chapter introduces DPOP, a special case of BTE that operates on a variable ordering which is given by a DFS arrangement of the problem graph. This guarantees that the restrictions from

Section 2.2 hold.

4.1 DPOP: A Dynamic Programming Optimization Protocol for DCOP

DPOP is a complete algorithm, and has the important advantage that it generates only a linear number of messages. This is important in distributed settings because sending a large number of small messages (like search algorithms do) typically entails large communication overheads.

In the following sections we will present in more detail DPOP's three phases. For a formal description, see Algorithm 4.

Algorithm 4 *DPOP: Dynamic Programming Optimization Protocol*

DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): each agent X_i does:

DPOP phase 1: DFS arrangement - run token passing mechanism as in Algorithm 3

1 At completion, X_i knows $P_i, PP_i, C_i, PC_i, Sep_i$

DPOP phase 2: UTIL propagation (bottom-up UTIL message propagation)

2 $JOIN_i^{P_i} = null$

3 **forall** $X_j \in C_i$ */* for all children of X_i; if X_i is a leaf, skip this */* **do**

4 wait for $UTIL_j^i$ message to arrive from X_j

5 $JOIN_i^{P_i} = JOIN_i^{P_i} \oplus UTIL_j^i$ *//we add to the join UTIL messages from children as they arrive*

6 $JOIN_i^{P_i} = JOIN_i^{P_i} \oplus R_i^{P_i} \oplus \left(\bigoplus_{X_j \in PP_i} R_i^j\right)$ *//also join all relations with parent/pseudoparents*

7 $UTIL_i^{P_i} = JOIN_i^{P_i} \perp_{X_i}$ *//use projection to eliminate self out of message to parent*

8 Send $UTIL_i^{P_i}$ message to P_i

DPOP phase 3: VALUE propagation (top-down VALUE message propagation)

9 wait for *VALUE*$^i_{P_i}(\langle Sep_i\rangle^*)$ msg from P_i *// $\langle Sep_i\rangle^*$ is the optimal assignment for all vars in $\langle Sep_i\rangle$*

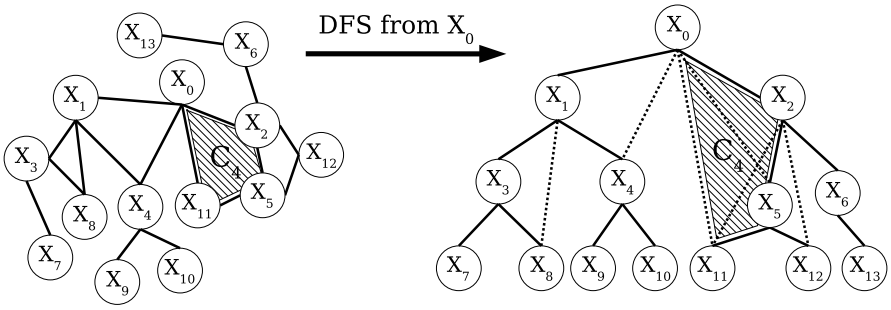
10 $X_i^* \leftarrow argmax_{v_i \in d_i}(JOIN_i^{P_i}[\langle Sep_i\rangle^*])$ *// slice $JOIN_i^{P_i}$ corresponding to $\langle Sep_i\rangle^*$; find best v_i*

11 **forall** $X_j \in C_i$ */* for all children of X_i; if X_i is a leaf, skip this */* **do**

12 send *VALUE*$(\langle Sep_i\rangle^* \cap \langle Sep_j\rangle \cup X_i^*)$ message to X_j

4.1.1 DPOP phase 1: DFS construction to generate a DFS tree

In phase 1, a **DFS traversal** of the graph is done using Algorithm 3. The DFS tree thus obtained serves as a communication structure for the other 2 phases of the algorithm: UTIL propagation (UTIL messages travel bottom-up on the tree), and VALUE propagation (VALUE messages travel top-down on the tree).



(a) a simple graph (b) a depth-first traversal from X_0

Figure 4.1: *A problem graph and a rooted DFS tree. Non-binary constraints like C_4 are treated as cliques of the variables involved. Tree edges are solid lines, while back-edges are dashed lines.*

4.1.2 DPOP phase 2: UTIL propagation

Phase 2 - **UTIL propagation**: this is a bottom-up process, which starts from the leaves and propagates upwards only through tree edges. In this process, the agents send *UTIL* messages (see Definition 14) to their parents. These messages summarize the influence of the sending agent and its whole subtree on the rest of the problem. They are equivalent to the induced constraints computed in the variable elimination steps in the bucket elimination scheme ([49, 51]).

Definition 14 (UTIL message) *$UTIL_i^j$, the UTIL message sent by agent X_i to agent X_j is a multi-dimensional matrix, with one dimension for each variable present in Sep_i. $dim(UTIL_i^j)$ is the set of individual variables in the message. Note that always $X_j \in dim(UTIL_i^j)$.*

The semantics of such a message is similar to an n-ary relation having as scope the variables in the context of this message (its *dimensions*). The size of such a message is the product of the domain sizes of the variables from the context.

Definition 15 (Slice) *Given a relation U (UTIL messages are relations) defined over a set of variables $dims(U)$, and an instantiated subset D of its dimensions ($D \subset dims(U)$), a **slice** through U along D, $U[D]$ is a lower-dimensionality relation S that has as dimensions $\{d|d \in \{dims(U) \setminus D\}\}$ and as values the values from U that correspond to the tuples $\{dims(U) \setminus D\}$. If $D = dims(U)$, $U[D]$ is a relation of arity 0, i.e. the corresponding value from U.*

Definition 16 (JOIN operator) *The $\oplus$ operator (join, or combine): $U = UTIL_i^j \oplus UTIL_k^j$ is the join of two UTIL matrices (relations). U is also a matrix (relation) with $dims(U) = dims(UTIL_i^j) \cup dims(UTIL_k^j)$ as dimensions. For each possible instantiation s of the variables in $dims(U)$, the*

corresponding value of $U[s]$ is the sum of the corresponding cells in the two source matrices: $\forall s \in U, U[s] = UTIL_i^j[s] + UTIL_k^j[s]$.

Example 6 *Given 2 matrices $UTIL_i^j$ and $UTIL_k^j$, with $dim(UTIL_i^j) = \{X_1, X_j\}$ and $dim(UTIL_k^j) = \{X_2, X_j\}$, then the value corresponding to $\langle X_1 = v_1^p, X_2 = v_2^q, X_j = v_j^r\rangle$ is $UTIL_i^j(X_1 = v_1^p, X_j = v_j^r) + UTIL_k^j(X_2 = v_2^q, X_j = v_j^r)$. Also, $dim(UTIL_i^j \oplus UTIL_k^j) = \{X_1, X_2, X_j\}$.*

Definition 17 (PROJECTION operator) *The $\perp$ operator (also known in the literature as* elimination *or* marginalization*): if $X_k \in dim(UTIL_i^j)$, $UTIL_i^j \perp_{X_k}$ is the projection through optimization of the $UTIL_i^j$ matrix along the X_k axis. Formally, $\forall s \in \{dim(UTIL_i^j) \setminus X_k\}, UTIL_i^j \perp_{X_k} [s] = max_{X_k} UTIL_i^j[s]$ (i.e. for each possible instantiation s of the variables other than X_k, the optimal instantiation for X_k is chosen and the corresponding utility recorded in $UTIL_i^j \perp_{X_k}$). The result $UTIL_i^j \perp_{X_k}$ is also a UTIL matrix, with one less dimension (X_k).*

The subtree of an agent X_i can influence the rest of the problem only through X_i's separator, Sep_i. Therefore, a message contains the optimal utility obtained in the subtree for each instantiation of Sep_i. Thus, messages are exponential in the separator size (bounded by the induced width).

To compute this message, an agent X_i has to join all the messages it received from its children, and the relations it has with its parent and pseudoparents, as in Equation 4.1:

$$JOIN_i^{P_i} = \left(\bigoplus_{X_c \in C_i} UTIL_c^i \right) \oplus \left(\bigoplus_{X_p \in \{P_i \cup PP_i\}} R_i^p \right) \tag{4.1}$$

To obtain its UTIL message, X_i projects itself out of the resulting hypercube as in Equation 4.2:

$$UTIL_i^{P_i} = JOIN_i^{P_i} \perp_{X_i} \tag{4.2}$$

Example 7 *In figure 4.1, X_4 computes its $UTIL_4^1$ message for X_1 as in equation 4.3:*

$$JOIN_4^1 = (\overbrace{\underbrace{UTIL_9^4 \oplus UTIL_{10}^4}_{dim=\{X_4\}} \oplus \underbrace{R_4^0}_{dim=\{X_4,X_0\}} \oplus R_4^1}^{dim=\{X_4,X_0,X_1\}}); UTIL_4^1 = \underbrace{JOIN_4^1 \perp_{X_4}}_{dim=\{X_0,X_1\}} \tag{4.3}$$

The leaf agents initiate the UTIL propagation. Subsequently, each agent X_i relays the UTIL messages as follows:

- Wait for *UTIL* messages from all children. Since all the respective subtrees are disjoint, joining messages from all children gives X_i exact information about how much utility each of its values

yields for the whole subtree rooted at itself. To assemble the message for its parent X_j, X_i has to also join R_i^j and any back-edge relation it may have with agents above X_j, as in Equation 4.1. Then it projects itself out of the result, as in Equation 4.2 (see lines 5-7 in Algorithm 4). The result is the $UTIL_i^j$ message (see equation 4.3 for $UTIL_4^1$).

- If X_i is the root agent, it receives all its *UTIL* messages as vectors with a single dimension (itself). It can then compute the optimal overall utility corresponding to each one of its values (by joining all the incoming *UTIL* messages) and pick the optimal value for itself (project itself out).

Remark 7 (Non-binary relations and constraints) *A k-ary relation/constraint is considered in the UTIL propagation only once, by being introduced in its UTIL message by the lowest agent in the DFS arrangement that is part of the scope of the relation. For example, in Figure 4.1(b), the constraint C_4 is introduced by agent X_{11} in its UTIL message to its parent, and subsequently propagated in the UTIL messages of agents X_5 and X_2. However, agents X_5 and X_2 do not explicitely take C_4 into account.*

4.1.3 DPOP phase 3: VALUE propagation

Phase 3 - **VALUE propagation** top-down, initiated by the root, when phase 2 has finished. Each agent determines its optimal value based on the computation from phase 2 and the *VALUE* message it has received from its parent. Then, it sends this value to its children through *VALUE* messages.

Clearly *DPOP* produces a linear number of messages. Its complexity lies in the size of the *UTIL* messages, which is time and space exponential in the width of the DFS ordering used.

Example 8 (A numerical example) *Figure 4.2 shows a simple example of a problem, to facilitate the understanding of the computation being performed by each agent. The problem has a tree structure (Figure 4.2(a)), with 3 relations $r_2^1(X_2, X_1)$, $r_3^1(X_3, X_1)$, and $r_1^0(X_1, X_0)$ detailed in Figure 4.2(b) and (c)-low.*

UTIL phase *X_2 and X_3 project themselves out of r_2^1 and r_3^1, respectively. The results are the green cells in r_2^1 and r_3^1 in Figure 4.2(b). The projections are the messages $UTIL_2^1$ and $UTIL_3^1$ that they send to X_1.*

X_1 receives the messages from X_2 and X_3, and computes the join $JOIN_1^0 = UTIL_2^1 \oplus UTIL_3^1 \oplus r_1^0$ - Figure 4.2(c). It then projects itself out: $UTIL_1^0 = JOIN_1^0 \perp X_1$; each value in the message represents the total utility of the entire problem, when X_0 takes that value. The result is depicted in Figure 4.2(d). X_0 receives this utility message from X_1, and can then simply choose its value that produces the larges utility for the whole problem: $X_0 = a$ ($X_0 = a$ and $X_0 = c$ produce the same result in this example, so either one can be chosen).

VALUE phase *The VALUE phase then starts. X_0 informs its child, X_1 of its choice via a message VALUE$(X_0 = a)$. X_1 then restores its value that was found optimal for $X_0 = a$: the blue cells in*

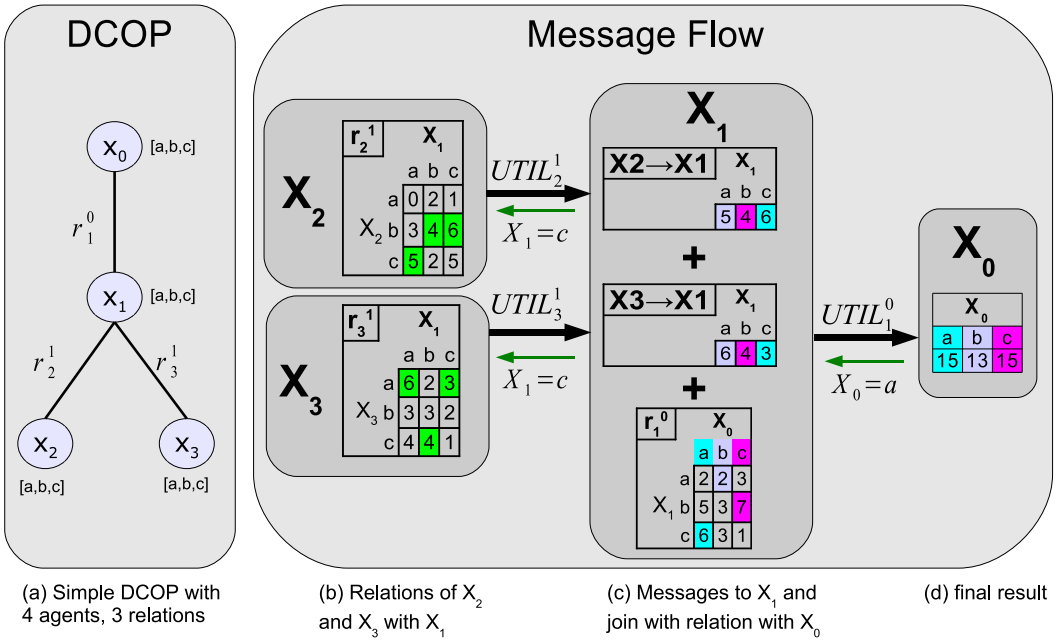


Figure 4.2: *A simple problem (a). Relations are detailed in (b) and (c)-low. Computation consists of (b)- projections of X_2 and X_3 out of their relations with X_1. Then, in (c) X_1 joins the messages from X_2 and X_3 with its relation with X_0. Finally, X_1 projects itself out, and sends the result to X_0 in (d).*

Figure 4.2(c) point to this computation, and X_1's optimal value is $X_1 = c$. The process continues with X_1 sending a message VALUE$(X_1 = c)$ to X_2 and X_3. Just like X_1 did, X_2 and X_3 restore their optimal values for $X_1 = c$, i.e. $X_2 = b$, and $X_3 = a$. The algorithm thus terminates with the optimal solution $\langle X_0 = a, X_1 = c, X_2 = b, X_3 = a\rangle$ that gives the maximal utility 15.

4.1.4 DPOP: Algorithm Complexity

Theorem 1 *DPOP (algorithm 4) requires a number of messages which is linear in the number of variables. The DFS construction and the VALUE propagation require messages of size linear in the number of variables. DPOP's complexity lies in the size of the UTIL messages, which are space-exponential in the induced width of the DFS tree used.*

PROOF. Follows easily from the complexity proof of BTE[107]. Specifically,

Number of messages: The DFS construction (algorithm 3) requires $2 * m$ messages, where m is the number of edges in the interaction graph. If n is the number of agents in the problem, then the UTIL phase requires $n - 1$ bottom-up messages, and the VALUE phase requires $n - 1$ top-down messages (one through each tree-edge).

Size of messages: By construction, both the DFS and the VALUE messages are of size linear in the number of agents in the problem. The BTE algorithm is time and space exponential in the size of the largest bucket encountered in the elimination process. In the case of DPOP, the size of each agent's bucket is given by the size of the agent's separator, and Proposition 7 shows that the size of the largest separator equals the induced width. □

4.1.5 Experimental evaluation

We performed experiments on meeting scheduling problems(MS)[127]. All experiments are run on a P4 machine with 1GB RAM, using the FRODO[154] simulation platform.

We generated a set of relatively large problems. The model is as in[127], and described in detail in Section 2.3.1. Briefly, an optimal schedule has to be found for a set of meetings between a set of agents. The test instances contained from 10 to 100 agents, and 5 to 60 meetings, yielding large problems with 16 to 196 variables. The larger problems were also denser, therefore even more difficult (induced width from 2 to 5).

The experimental results are presented in Figure 4.3. Figure 4.3(a) shows the number of messages exchanged, and Figure 4.3(b) shows the sum of all message sizes, in bytes. Figure 4.3(c) shows the runtime in milliseconds. [1]. Please notice the logarithmic scale! ADOPT did not scale on these problems, and we had to cut its execution after a threshold of 2 hours or 5 million messages, whichever occured first. The largest problems that ADOPT could solve had 20 agents (36 variables).

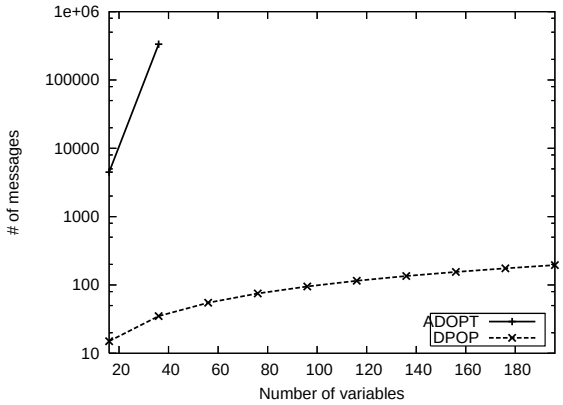
As predicted by the theory, DPOP only requires a linear number of messages. What is interesting to note is that even though DPOP sends larger messages than ADOPT, overall, it exchanges much less information (Fig 4.3(b)). We believe there are 2 reasons for this: ADOPT sends many more messages, and because of its asynchrony, it has to attach the full context to all of them (which produces extreme overheads).

4.1.6 A Bidirectional Propagation Extension of DPOP

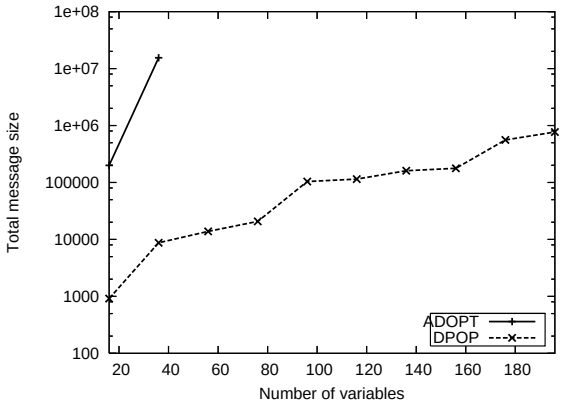
In DPOP, any UTIL message from an agent to its parent summarizes the utility information from all the subtree rooted at the respective agent. Therefore, the bottom-up UTIL propagation gives the root global utility information, but all other agents have accurate *UTIL* information only about their subtrees.

Similar to BTE[107], we extend the *UTIL* propagation by making it *bidirectional*, in the sense that it traverses the DFS tree in both directions: not only bottom to top, as in DPOP, but also top to bottom, from each agent to its children. A message from a parent to its child summarizes the utility information from all the problem except the subtree of that child. This new message can be joined together with all

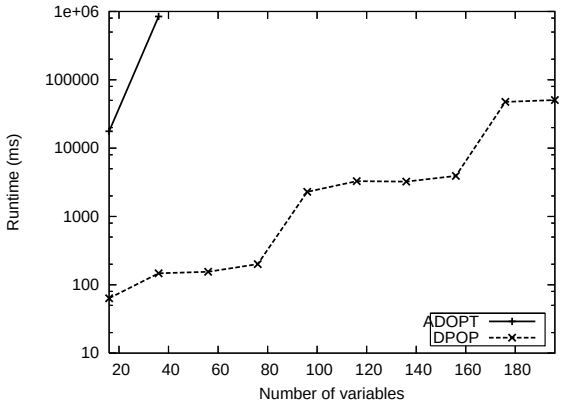
[1] Each data point is an average over 10 instances



(a) Number of messages



(b) Total size of messages



(c) Runtime in miliseconds

Figure 4.3: DPOP vs ADOPT - evaluation on meeting scheduling problems.

the messages received by an agent from its children. The result is a summary of the utility information from the whole problem, which gives each agent a global view of the system, logically making each agent in the system equivalent to the root.

Notice that a similar effect can be obtained by running DPOP n times, once with each variable as the root. However, this approach clearly would require spending more effort than the bidirectional utility propagation we propose here: roughly speaking, n times the effort spent by DPOP, vs. twice this effort.

The process is initiated by the root when it has received the *UTIL* messages from its children. Each agent X_i (including the root) computes for each of its children X_j a $UTIL_i^j$ message. To do so, X_i first builds the join of the messages received from its other neighbors than X_j, plus the *relation* it shares with X_j: $JOIN_i^j = R_i^j \oplus \left(\bigoplus_{c \in \{P_i \cup C_i \setminus X_j\}} UTIL_c^i\right)$.

The set of dimensions of the joined message is always a superset of the dimensions that have to be passed down to the children. Subsequently, agent X_i applies a projection step to the outgoing message for X_j, such that only the *relevant* dimensions are kept. This is done by projecting out in principle all dimensions not present in Sep_j, with two exceptions:

1. the dimension of X_j itself
2. the dimension of the sending agent X_i, *if* X_i has a pseudochild in the subtree rooted at X_j; this information is a byproduct of the DFS algorithm.

Once X_i has determined the relevant dimensions, it projects out everything else:

$$UTIL_i^j = JOIN_i^j \perp_{X_k \in \{dim(JOIN_i^j) \setminus dim(UTIL_i^j)\}}$$

Example 9 (Bidirectional UTIL propagation) *Let us consider the problem from Figure 4.4 (same DFS as in Figure 4.1). As a result of the normal bottom-up UTIL propagation, X_0 receives the $UTIL_2^0$ message from its child X_2 and can now compute its UTIL message for X_1: $JOIN_0^1 = UTIL_2^0 \oplus R_0^1$. X_0 has a pseudochild (X_4) in the subtree rooted at X_1, therefore it cannot project itself out of the UTIL message it sends to X_1. Therefore, X_0 sends to X_1 $UTIL_0^1 = JOIN_0^1$.*

Subsequently, X_1 builds $JOIN_1^3 = R_1^3 \oplus UTIL_0^1 \oplus UTIL_4^1$. As $UTIL_3^1$ previously received by X_1 from X_3 does not contain X_0 as a dimension, X_1 will project X_0 out of the UTIL message it will send to X_3. Similarly to X_0, X_1 also identifies a backedge to itself originating from the subtree rooted at X_3. Therefore, it cannot project itself out of the message for X_3: $UTIL_1^3 = JOIN_1^3 \perp_{X_0}$.

X_1 then prepares its message for its other child, X_4: $JOIN_1^4 = R_1^4 \oplus UTIL_0^1 \oplus UTIL_3^1$. As $UTIL_4^1$ previously received by X_1 from X_4 does contain X_0 as a dimension, X_1 will not project X_0 out of the UTIL message it will send to X_4. Furthermore, X_1 does not have any backedge with any agent in the subtree rooted at X_4, so it can project itself out. Thus, X_1 sends X_4 $UTIL_1^4 = JOIN_1^4 \perp_{X_1}$.

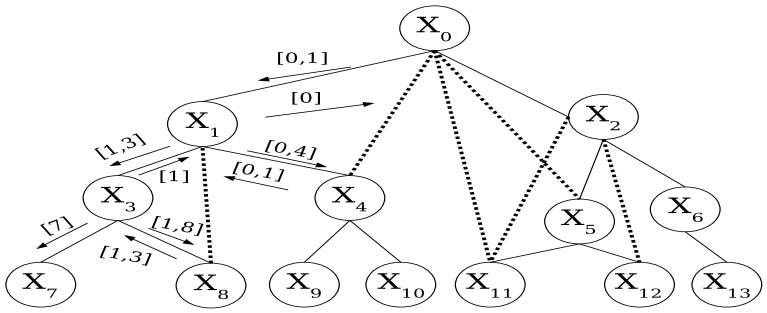


Figure 4.4: *An example of a problem where bidirectional propagation is performed. Each arrow represents an UTIL message, and the numbers in brackets above represent the dimensions of the UTIL message. For example, $UTIL_0^1$ has two dimensions: X_1 and X_0 (because of the backedge R_0^4, X_0 cannot project itself out from the message going to X_1).*

Chapter 5

H-DPOP: compacting UTIL messages with consistency techniques

DPOP groups many valuations together in fewer (and also larger) messages, thus producing small communication overheads. However, the maximum message size is always exponential in the induced width of the constraint graph, leading to excessive memory and communication requirements for problems with large width.

Many real problems contain hard constraints that significantly reduce the space of feasible assignments. However, dynamic programming does not take advantage of the pruning power of these hard constraints; thus, DPOP sends messages that explicitly represent all value combinations, including many infeasible ones. Search algorithms mitigate this problem by various methods for pruning (partial assignments that have lead to an inconsistency are not further explored). Further pruning is achieved through consistency techniques, as well as the branch-and-bound principle.

This chapter brings two contributions: the first is H-DPOP, a hybrid algorithm that is based on DPOP. H-DPOP uses Constraint Decision Diagrams (CDDs, see [34]) to rule out infeasible combinations, and thus compactly represent UTIL messages. For highly constrained problems, CDDs prove to be extremely space-efficient when compared to the extensional representation used by DPOP: experimental results show space reductions of more than 99% for some instances. H-DPOP is an orthogonal technique, which can nicely complement other improvements to DPOP like MB-DPOP, LS-DPOP, A-DPOP, etc.

The second contribution of this chapter is a detailed comparison between search with caching[32, 42, 132] and dynamic programming with CDDs. H-DPOP outperforms the search algorithm by a large margin on the number of messages exchanged while exploring a similar search space and thus is better suited for distributed environments.

In this chapter, we consider how to apply known hard constraints on feasible value combinations to prune such combinations, so that only information that actually corresponds to feasible solutions is transmitted. We do this by encoding value combinations using *constrained decision diagrams*(CDDs)

([34]). CDDs eliminate all inconsistent values and only include costs or utilities for value combinations that reach a consistent leaf node. In experiments on several practical problems, we show that this cuts message size by up to 99%, putting problems of practical size within reach of H-DPOP.

A technique that explores hard constraints in a similar way is to cache partial results during the search [42], as implemented in the NCBB algorithm ([33]). Similar to dynamic programming with CDDs, the caches contain only utility values for value combinations that are actually consistent. However, the pruning carried out by CDDs is very different from that achieved by backtrack search: while backtrack search prunes all value combinations that are inconsistent with variables that are *higher* in the ordering, CDDs do the pruning from the bottom up and prune value combinations that are inconsistent with variables *lower* in the ordering.

To compare the pruning achieved by the two methods, we have modified the NCBB search algorithm ([33]) to obtain another version that (a) maintains a complete cache and (b) does not use the branch-and-bound heuristic which we cannot reproduce in CDDs. We compare the space explored in dynamic programming with CDDs to that explored in backtrack search by comparing the size of the cache that has been used. We evaluate our CDD-based algorithm against different versions of NCBB, and show on several example domains that CDDs achieve essentially the same pruning achievable by search with the added advantage that only a linear number of messages are required. Thus, dynamic programming with CDDs achieves similar benefits but is more suitable for distributed settings.

The rest of this chapter is structured as follows: Section 5.1 presents an example problem which contains hard constraints, and introduces constraint decision diagrams (Section 5.1.1). Section 5.2 introduces the H-DPOP algorithm. Section 5.3 discusses search in general, and compares H-DPOP with the NCBB algorithm with caching from a theoretical point of view. Section 5.4 contains a comprehensive experimental evaluation of H-DPOP against DPOP and NCBB. Section 5.5 places H-DPOP in the context of existing work, and Section 5.6 concludes.

5.1 Preliminaries

Without loss of generality, hard constraints can be simulated using soft constraints by assigning utility $-\infty$ to disallowed tuples, and utility 0 to allowed tuples. Then, simply using any utility maximization algorithm such as DPOP avoids infeasible assignments and finds the optimal solution. However, by doing so one does not take advantage of the pruning power of hard constraints. This drawback becomes severe for difficult problems (high induced width).

We introduce below one such real world problem and show the space reduction ability of hard constraints.

Optimal query placement: Recall from Section 2.3.3 the problem of optimally placing a set of query operators in an overlay network. Each user wants a set of services to be performed by servers in the network. Servers are able to perform services with distinct network and computational characteristics. Each server receives hosting requests from its users (together with the associated utilities). We model the resulting DCOP with servers as variables (agents) and the possible service combinations as the domains.

To avoid accounting the utility from the same service being placed simultaneously on two servers we introduce hard constraints between server pairs. These constraints disallow the same service to be executed by two servers at a time. Although this constraint is simple, it makes the problem highly constrained and computationally difficult.

Note that the above model may not be an exactly equivalent model for optimal query placement but it helps to make the problem tractable. The optimal solution may include running a service on more than one server but the problem would become much more complex in its originality.

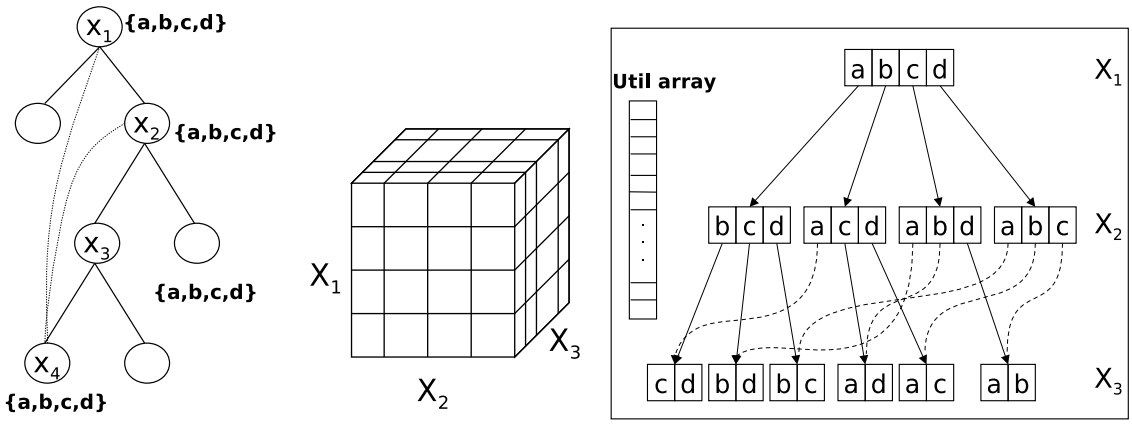
Figure 5.1(a) shows a DFS tree arrangement for servers in an overlay network. The services each server can execute are listed adjacent to nodes. During the utility propagation phase of DPOP node X_4 will send a hypercube with X_1, X_2 and X_3 as context variables to its parent X_3 (see figure 5.1(b)). However such a message scheme will send combinations which will never appear in a valid solution. For example combinations like $\langle X_1 = a, X_2 = a, X_3 = b\rangle$ which share a common service are infeasible. The total size of this hypercube will be 64 (4^3) with only 24 ($4!$) valid combinations. Eliminating these combinations using hard constraints can provide significant savings.

Consider an instance of a server problem with 9 variables (servers) with the same domain of size 9. The resulting network will be a chain with constraints between every server pair. The maximum size of hypercube in DPOP will be 9^9 and the number of valid combinations will be only $9!$. So we are wasting 99.9% of the space in the message by sending irrelevant combinations. With the help of hard constraints we can prune such infeasible combinations and get extreme savings.

5.1.1 CDDs: Constraint Decision Diagrams

CDDs (constrained decision diagrams)[34] are compact representations for general n-ary constraints. They generalize binary decision diagrams (BDD)[28]. Their main feature is that they combine constraint reasoning and consistency techniques with a compact data structure. Unlike extensional representations that store each individual tuple separately (therefore requiring memory exponential in the arity of the constraint), CDDs have the potential to drastically reduce space requirements.

Formally, a CDD is a rooted, directed acyclic graph (DAG) $G = (V \cup T, E)$. The $0-terminal$ $(0 \in T)$ represents $false$ and $1 - terminal$ $(1 \in T)$ represents $true$. Each non terminal node $v \in V$ connects to a subset of nodes $U \subseteq V \cup T - \{v\}$. It is denoted by a non-empty set $\{(c_1, u_1), ..., (c_m, u_m)\}$.



(a) DFS Tree (b) Hypercube (c) CDD Message

Figure 5.1: *A DFS tree, an extensional UTIL message (hypercube) sent from X_4 to X_3 and its CDD equivalent. The constraints on X_1, X_2, X_3, X_4 impose that they must take different values. The hypercube contains all combinations of X_1, X_2, X_3, whereas the CDD only the feasible ones, thus saving space.*

Each branch (c_j, u_j) consists of a constraint $c_j(x_1, ..., x_k)$ and a successor u_j of v.

A CDD rooted at the node $v = \{(c_1, u_1), ..., (c_m, u_m)\}$ is $reduced$ if and only if each CDD graph G'_j rooted at u_j is either terminal or reduced, and

$$c_i \wedge c_j \equiv false \tag{5.1}$$

$$u_i \not\equiv u_j \tag{5.2}$$

Example 10 (CDD) *We show in Figure 5.1(c) an example CDD that represents compactly a ternary constraint (C) between X_1, X_2 and X_3 with domain (D_i) listed adjacent. The constraint C requires the variables to take distinct values. Each CDD node is of the form $\{(x_k \in r_1, u_1), ..., (x_k \in r_m, u_m)\}$ where $r_1, ..., r_2 \subset D_k$ are pairwise disjoint to satisfy Property (1) of a reduced CDD. Property (2) allows node sharing marked by dashed lines in Figure 5.1(c).*

5.2 H-DPOP - Pruning the search space with hard constraints

The H-DPOP algorithm combines this pruning power with CDDs to effectively reduce message size. Figure 5.1(c) shows the corresponding CDD message for the hypercube X_4 sends to X_3 in Figure 5.1(a). In a CDD, every path from root to leaf is a valid combination of domain values of the involved variables. The explicit representation of domain values and an insight into the problem nature allows us to prune combinations like $< (X_1, a)(X_2, a)(X_3, b) >$ for service placement problems even

at the bottommost level.

Utilities in a CDD message are represented by a linear array storing utility values indexed by path numbers. Each path in a CDD is assigned a unique index obtained by a DFS traversal of the CDD tree. Additionally we also need to send the domain values of each variable in a CDD message. This step is necessary to ensure pruning at higher levels which is based on examining combinations of actual domain values.

Definition 18 *$Dim(X)$ is the tuple $< X, dom(X) >$ consisting of the variable X and its domain $dom(X)$.*

We now describe the **CDDMessage** which node X will send to P_X. It is composed of three components:

- *CDDTree* : It represents all valid combinations of variables involved in the message. Each level in *CDDTree* corresponds to one variable.
- *UtilArray*: It is the array of all utilities corresponding to each path in *CDDTree.*
- *DimensionArray* : It is an array containing $Dim(X_i)$ where $X_i \in \{$variables involved in message$\}$.

As in DPOP, H-DPOP contains three phases as well: DFS arrangement, bottom-up UTIL propagation and top-down VALUE propagation. The DFS and VALUE phases are identical to the ones of DPOP, and the modified UTIL propagation phase is described below in Section 5.2.1.

5.2.1 UTIL propagation using CDDs

This phase is similar to the UTIL phase of DPOP, with the difference that the extensional representations of UTIL messages from DPOP (hypercubes) are replaced with CDD messages and the associated utility vectors. The JOIN and PROJECT operators on hypercubes are redefined in the following for CDD messages.

5.2.1.1 Building CDDs from constraints:

Algorithm 5 describes the construction of CDDTree corresponding to the Hypercube with dimension set *Dim[dimSize]. C* is the partial assignment currently found to lead to a valid solution. Whenever a new domain value is added to *C*, a consistency check is performed in line 6 to see if the newly instantiated domain value will lead to a solution. This is a key step in pruning the search space as

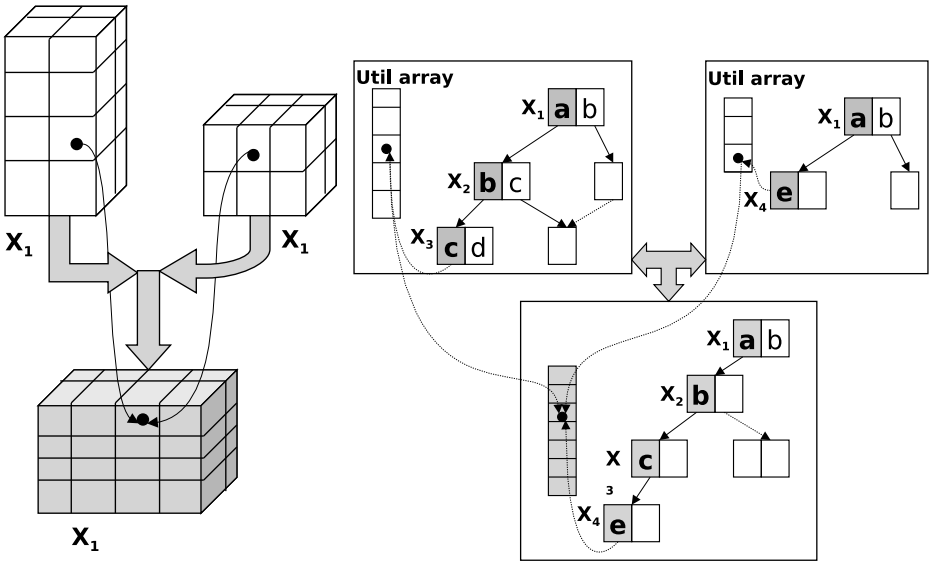
Algorithm 5 *Construction of a CDDtree*

Procedure `ConstructCDD`
input : Dim*[dimSize], C[dimSize], currentLevel*
output : The root of the CDDTree

1 **if** *dimSize == 0* || *currentLevel == dimSize* **then return** *null*
2 $X_k =$ node at currentLevel, D_k = X_k.domain
3 $S = \emptyset$, $D'_k = \emptyset$
4 **forall** $d \in D_k$ **do**
5 $C[currentLevel] = d$
6 **if** `isConsistent(`*C, currentLevel*`)` *== true* **then**
7 u = `ConstructCDD(`$Dim, C, currentLevel + 1$`)`
8 **if** $u ==$ *null* || $(u \neq null \ \cap \ u \neq 0)$ **then**
9 $S = S \cup \{< d, u >\}$
10 $D'_k = D'_k \cup \{d\}$
11 **if** $D'_k = \emptyset$ **then return** *0*
12 v = `mkNode(`X_k, S, D'_k`)`
13 **return** v

Procedure `mkNode`
input : X_k, *S*, D'_k where X_k = variable, S=children, D'_k= valid values
output : A CDDNode corrresponding to variable X_k with given domain and children set

14 $v = \{(X_k \in r, u) : d, d' \in r \Longleftrightarrow < d, u >, < d', u > \in S\}$
//`i.e.` $X_k \mapsto d \ and \ X_k \mapsto d' \ point \ to \ same \ node \ u$
15 **if** *htable.*`get(`v.`hashKey())` $== null$ **then**
16 $htable$.`add(`*v.hashKey(), v*`)` *//htable contains all discovered nodes*
17 **return** *v*
else
//`i.e.` $\exists v' \quad s.t. \ v' \equiv v$
18 **return** v'



(a) Hypercube Join (b) CDDMessage join

Figure 5.2: H-DPOP: comparative view of joining hypercubes vs. joining CDDs

inconsistent combinations are ruled out via this check. Parameter *currentLevel* denotes the current level in CDDTree under exploration. Its initial value is zero denoting the CDDroot.

The procedure `ConstructCDD` is based on a depth-first backtrack search algorithm (see [34]). Set S (initially empty) consists of the branches of the CDDNode, and D'_k consists of values of variable X_k which can lead to valid combinations. Next, for each value in the domain of X_k (=D_k), we check if it can lead to a feasible solution (line 6). If no, it is ruled out otherwise we recursively invoke `ConstructCDD` to find the CDDNode *u* for the next level (*currentlevel+1*, line 7). If *u* is a 1-terminal (*null node*) or is not a 0-terminal , we add the branch (*d,u*) to *S*, and insert d to D'_k (lines 8 to 10). If $D'_k = \emptyset$ after all iterations are over, a 0-terminal is returned. Otherwise, `mkNode` is called to return the CDDNode for the current variable with given children and domain set (S and D'_k respectively).

Procedure `mkNode` is shown in algorithm 5. In line 14, an intermediate node *v* is created such that for every $d \in r$, $X_k \mapsto d$ leads to the same child node *u*. Next we check if an equivalent node $v\prime$ exists for node v to satisfy property (2) of a reduced CDD (line 15). If an equivalent node exists we reuse that node otherwise insert *v* to *V* and return *v* (line 17).

5.2.1.2 Implementing the JOIN operator on CDD messages

In Algorithm 6 we describe the method for combining two CDD messages. The extra parameter *leafDimension* specifies which variable should be placed at the leaf level in the resulting combined CDDTree

(line 2). Each node places itself as the *leafDimension* while combining CDD messages from its children. This is an optimization which ensures that each node can project itself out of the resulting CDD message very efficiently (see function `projectMine`, algorithm 7) before sending the message to its parent. The union of dimensions of combining messages forms the dimensions of combined message (line 1). With this new set of dimensions, a new CDD message is constructed with an empty *UtilArray*. The for loop in line 5 iterates over all the paths of the newly formed CDDTree, finds the relevant contributions from individual CDD messages (function `findUtil`, line 7), and sets the utility of the current path in the combined message (line 9). Finally, the combined CDD message is returned after setting the utility of each path. Figure 5.2 shows the join of hypercubes and CDDs.

The procedure `findUtil` (algorithm 6) returns the utility value corresponding to CDDMessage's local contribution for the input source path with the specified set of dimensions *unionDim*. The Array *myPath* stores the local contribution of the CDDMessage to input *srcPath*. It is initialized with values from *srcPath* for the corresponding dimensions in *myDim* and *unionDim* (line 13). The utility value for *myPath* is extracted from *UtilArray* by finding the index of this path (line 14). To increase performance, each CDDMessage hashes every path of its CDDTree with value as path index and key as the path itself.

5.2.1.3 Implementing the PROJECT operator on CDD messages

Procedure `projectMine` (algorithm 7) is used by a node to project out its own dimension after it has joined the UTIL messages from its children and the relations with its parent/pseudo parents. Since the `combineCDDMessages` function places the dimension of the current node as the leaf level of CDDTree, projecting out the current node is very efficient: we iterate through all the paths (line 2) and choose the best utility among paths having the same prefix, except for the leaf level (line 7). We also need to reconstruct the CDD message and initialize the utility array after the projection operation to keep the CDD size optimal. Finally the newly formed CDDMessage is returned to be sent to the parent of the current node.

5.2.1.4 The `isConsistent` plug-in mechanism

The `isConsistent` (see algorithm 5, line 6) function is like a gateway to the constraint problem being solved and uses hard constraint propagation for pruning the search space. Until now existing DCOP algorithms like ADOPT or DPOP did not try to take advantage of domain specific knowledge while solving a particular DCOP instance. H-DPOP is unique in this sense as it provides the constraint optimization algorithm with knowledge about the problem domain through this modular plug-in mechanism. Our results show that this knowledge can help reducing the size of the UTIL messages by up to 99%. This function is problem-specific and encapsulates the pruning logic into the H-DPOP algorithm. The input to this function is the constraint array C, which is a partial assignment. The function then processes this input using hard constraint propagation and determines if C represents a feasible combi-

Algorithm 6 *Combining two CDDMessages: JOIN operation*

Procedure `combineCDDMessages`
input : *Msg1, Msg2, leafDimension*
output : Combined CDDMessage of Msg1 and Msg2

begin

1 Dim[] *union* = $Msg1.DimensionArray \cup Msg2.DimensionArray$
2 Rearrange *union* array to make $leafDimension$ as the last one
3 *CDDRoot* = `ConstructCDD(`*union, new Array(union.length), 0*`)`$\cap\, each\ path \in \{Msg1 \cup Msg2\}$
4 *combinedMsg* = new `CDDMessage(`*CDDRoot, union, CDDRoot.pathsCount*`)`
//`pathsCount represents total paths from root to leaves`

end

5 **foreach** *path of CDDTree with root = CDDRoot* **do**

6 *path* = current path under consideration
7 *util1* = Msg1.`findUtil(`*union, path*`)`
8 *util2* = Msg2.`findUtil(`*union, path*`)`
9 *combinedMsg*.`setUtility(`*util1+util2, path.index*`)`

10 **return** *combinedMsg*

Procedure `findUtil`
input : *unionDim, srcPath*
output : The utility value corresponding to local contribution to *srcPath*

11 myDim = this.*DimensionArray*
12 Initialize *myPath* = new Vector(myDim.*length*)
13 *myPath* =
$< (d_i = srcPath[j]) >:\ i \in [0, myDim.length] \cap \exists j\ s.t.\ srcDim[j].id = myDim[i].id$
14 *index* = htable.`getvalueByKey(`*myPath*`)`
15 **return** *this.UtilArray[index]*

Algorithm 7 *PROJECT operation for a CDDMessage*

Procedure `projectMine`: projects out the last dimension of this CDDMessage
output : returns the new CDDMessage

```
1  Initialize BestUtilities = new Vector()
2  foreach path of CDDTree of this CDDMessage do
3      path = currentPath under consideration
4      pathPrefix = path.prefix(0, path.size-1)
5      if utility already set for pathPrefix then
6          continue
       else
7          util = Max(P_i.util : P_i.prefix(0,P_i.size−1) = pathPrefix ∩ P_i ∈ {paths of CDDTree})
8          BestUtilities.set(util, pathPrefix)
9  Initialize newDim[] to this.DimensionArray[0] to [totalSize-1]
10 newTree = constructCDD(newDim, new Array(newDim.length), 0)
11 newMsg = new CDDMessage(newTree.root, newDim, newTree.pathsCount)
12 Initialize newMsg.UtilArray from BestUtilities
13 return newMsg
```

nation. For the server problem described in Section 5.1, `isConsistent` simply returns `false` for all partial assignments where several variables take the same value, because the hard constraints do not allow this.

Procedure 8 isConsistent(C, currentIndex)

output : true if C is valid, false otherwise
for $i = 0$ ***to*** $currentIndex - 1$ **do**
    if $C[i] == C[currentIndex]$ **then return** false
return true

The next section discusses the relationship between H-DPOP and search algorithms.

5.3 Comparing H-DPOP with search algorithms

Distributed search ([33, 81, 141, 194]) is an alternative approach to inference based algorithms like DPOP. Algorithms based on sequential search naturally provide pruning based on hard constraints: partial assignments that have led to an inconsistency are not further explored, and the search backtracks. Further pruning is achieved through more sophisticated consistency techniques, and by using variants of the branch-and-bound principle. The main advantage of search over inference based algorithms like DPOP ([160]) is that search uses only polynomial space, which makes it suitable for memory limited platforms. The main drawback is that typically, search algorithms require an exponential number of

small messages, thus producing high network overheads.

A major improvement to search has been to use a cache at each node to remember past results ([42]). The total size of all caches represents the explored search space in sequential search. Experimentally we will show that the total explored search space in search with *full caching* is similar to the explored space in H-DPOP. This comparison will provide a further testimony to the pruning power of H-DPOP with an important advantage that H-DPOP uses linear number of messages, as opposed to an exponential number in search.

In addition, we will show comparisons with a version of search which exploits only hard constraints without using the branch and bound principle. This version of search is closer to our H-DPOP algorithm as H-DPOP does pruning using hard constraints only without any other bounding. Our experiments show that search only using hard constraints provides similar performance as branch and bound search with only minimal degradation in cache size and message exchanges. This further highlights that the hard constraints are the dominating factor in all these problems and H-DPOP which efficiently exploits them with linear number of messages is superior to search.

5.3.1 NCBB: Non Commitment Branch and Bound Search for DCOP

NCBB ([33]) is a polynomial space distributed branch and bound search for distributed optimization. The basic idea of branch and bound is the same as in centralized branch and bound search ([118]). The distributed nature of search allows it to use different agents to search the non intersecting parts of search space concurrently providing speed and computational resource advantage. It also allows for eager propagation of bound changes from children to parent providing better pruning. The details of NCBB can be found in [33]. We will describe it shortly here.

NCBB works on the DFS tree arrangement of agents in the constraint graph. The DFS ordering can be done in the same way as in Section 3.4.1 or in[33]. The main advantage of such an ordering over the traditional OR based search is that given ancestor assignments the agents in a given subtree can work *independently* to minimize their cost. The time complexity of $O(d^n)$ in the OR based search (d is the maximum domain size, n is the number of agents) reduces to $O(bd^{H+1})$, where b is the branching factor of DFS tree arrangement, d is the maximum domain size and H is the depth of DFS traversal of constraint graph.

During the initialization of search in NCBB all agents compute the global upper and lower bounds on the solution cost. Then each agent chooses its value greedily provided the ancestor assignments to minimize its contribution (excluding the its subtree) to the global solution cost. After initialization agents start performing the main search procedure. An agent X_i initiates search (in its subtree) only after receiving an explicit *SEARCH* message from its parent. Before this *SEARCH* message all X_i's ancestors choose their values and announce them to all their descendants.

A distinct feature of NCBB is that when X_i selects a value to explore on a given subtree it may choose different values for its different subtrees (the *non commitment part*). The advantage of such concurrent search is that it allows for tighter upper bound when a value is used in a different subtree as we can take into account the already known cost for the completed subtree searches and do better pruning. Once the search is finished at the root for all subtrees and for each of root's values we have the solution to the optimization problem.

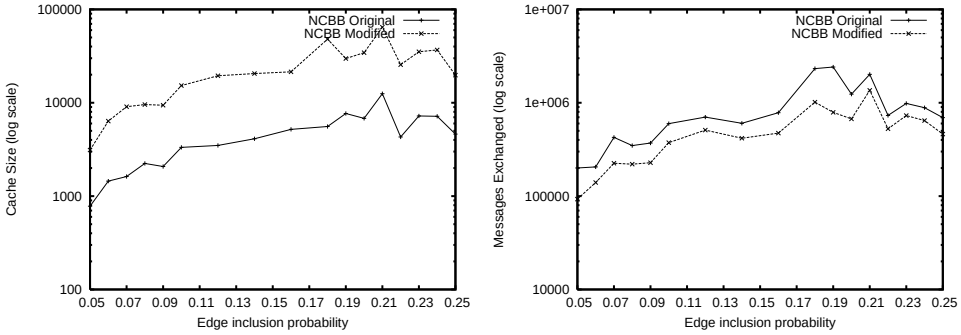
5.3.1.1 NCBB with caching

The main advantage of search over inference based algorithms like DPOP ([160]) is that search uses only polynomial space making it suitable for memory limited platforms. However the polynomial space comes with a price: search forgets everything from the past, so it may have to re-explore some parts of the search space. A natural extension of search would be to use a variable sized cache at each agent storing the previous search results, so that when the search explores previously visited search space its value can be directly looked upon in the cache. Such a scheme greatly improves the performance of search (as shown in[42], [32]) and allows the user to control the space-time tradeoff by varying the cache size (using user defined cache factor).

An advanced version of NCBB ([32]) incorporates such a caching scheme. The maximum cache size at any node X_i is $d^{|Sep_i|}$ (see Section 3.1.2.3). In the original NCBB the cache stores the solution cost, indexed by the value assignment in Sep_i, provided by the subtree at X_i. The results are entered in the cache *given* the subtree can provide a solution within the current bounds at X_i. Otherwise the result is not cached. In practice such a scheme keeps the cache size smaller at the expense of extra effort (number of messages) invested for re-exploring the previously pruned and not cached parts of the search space.

$NCBB^*$: NCBB with a modified caching policy We have modified the caching in NCBB (called $NCBB^*$, shown in graphs as *NCBB Modified*) so that we also store the $< cost, Sep_i >$ pair at X_i even if for the current Sep_i assignment subtree can not provide a solution within the bounds. This saves us the extra effort when such a combination is encountered again in the search. $NCBB^*$ works on the same DFS tree as in H-DPOP using the MCN heuristic ([127, 207]) to provide comparable results. We have implemented another version of search which works only on the hard constraints without any other bounds is *NCBB Hard Constraints*.

In $NCBB^*$ we store the $< Sep_i, cost >$ pairs in the cache even if the subtrees rooted at agent X_i can not provide a solution within the current bounds. This modification saves the overhead in terms of messages exchanged when the same Sep_i assignment is explored again. Figure 5.3(b) shows the number of messages exchanged in NCBB and $NCBB^*$ (for experimental setup see Section 5.4.1.2, here it suffices to know that p or edge inclusion probability, plotted on the x axis, is a graph parameter).



(a) Cache Size (b) Messages Exchanged

Figure 5.3: NCBB vs $NCBB^*$: NQueen graphs

$NCBB^*$ always requires a smaller number of messages, and the savings are often quite significant (between 40% and 65% for p=0.19).

In contrast, the cache size used in $NCBB^*$ is more than the cache in NCBB but this increase in the cache is more than compensated by reduced message exchange. It is better to have a slightly bigger local cache than to increase the network overhead by exchanging larger number of messages. The idea of search with caching being better than the search alone is a testimony to this approach.

Hence in our opinion the comparison of H-DPOP with $NCBB^*$ is more accurate than H-DPOP vs NCBB, as both H-DPOP and $NCBB^*$ traverse a similar search space.

5.3.2 Comparing pruning in search and in H-DPOP

NCBB Hard Constraint and H-DPOP both prune the search space based on only hard constraints, so we would expect that the size of explored search space should be identical in both cases. However, the experimental results show that there are slight variations. We show in the following the different pruning strategies employed by search and H-DPOP which account for this difference.

Figure 5.4 shows a DFS arrangement of a constraint network. First consider the pruning done by H-DPOP. H-DPOP does pruning from *bottom up*. As the message goes up from the leaf in the subtree of node N_3 it prunes all the inconsistent combinations. H-DPOP always explore the search space at any node N_i which is consistent given the assignment of Sep_i nodes at N_i and the subtree of N_i. On the contrary this is not true for search algorithms. Search prunes assignments from top to bottom. At the node N_1, the partial solution from root N_{root} until N_1 is consistent. However, there is no guarantee that this consistent partial solution will be consistent upon further exploration of the subtree at N_1.

Furthermore, there is no guarantee in either search or the H-DPOP algorithm that they always ex-

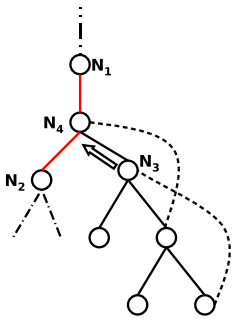


Figure 5.4: A DFS tree arrangement to illustrate differences in bottom-up pruning (H-DPOP) vs. top-down pruning (search)

plore only *globally consistent* combinations. H-DPOP explores consistent solutions in the assignment space of Sep_i nodes and the subtree of any node N_i. However, such combinations may become inconsistent as UTIL messages goes up the DFS tree on two accounts. In Figure 5.4, N_3 sends its message to its parent. The parent combines this message message with its other child N_2's message. During this process the combinations which are present in both sibling's messages are passed up, and the rest are pruned. The other source of pruning are the constraints of N_4 with its parent and pseudo parents, which could make some combinations from the children of N_4 inconsistent.

In search, any consistent partial solution may become inconsistent as the search expands lower nodes in the DFS tree. So this leads to inconsistent search space exploration in *NCBB Hard Constraint*.

5.4 Experimental Results

This section discusses the performance of H-DPOP on a number of problems: optimal query placement(introduced in Section 5.1), distributed graph coloring and winner determination in distributed combinatorial auctions (only with buyers). All these problems have a satisfaction component (solutions must not violate any hard constraints, thus incurring infinite costs), and an optimization one (maximizing utility, or minimizing cost, respectively). The experiments were performed on the FRODO platform (publicly available,[154]). The machine used has 1GB RAM with two P4 3GHz processor.

We performed two sets of experiments : (1) H-DPOP vs DPOP (see Section 5.4.1) and (2) H-DPOP vs NCBB (see Section 5.4.2). The H-DPOP vs DPOP experiments mainly focus on the space savings provided by H-DPOP by pruning the search space. The second set of experiments (H-DPOP vs NCBB) compares the search space explored and message exchanges in H-DPOP vs different versions of NCBB. For space comparisons we compare the logical sizes of the corresponding units (hypercubes in DPOP, CDDs in H-DPOP and total cache size in search with caching).

5.4.1 DPOP vs H-DPOP: Message Size

These experiments mainly focus on the space savings provided by H-DPOP by pruning the search space. We have performed 4 sets of experiments: query placement problems, graph coloring problems, n-queens problems, and combinatorial auctions problems.

5.4.1.1 Optimal query placement in an overlay network

For experiments the problem is made deliberately very constrained by assuming that each server is able to execute the complete set of services. For simplicity's sake, each server can execute only a single service at a time. The objective of the DCOP algorithm is to maximize the overall utility.

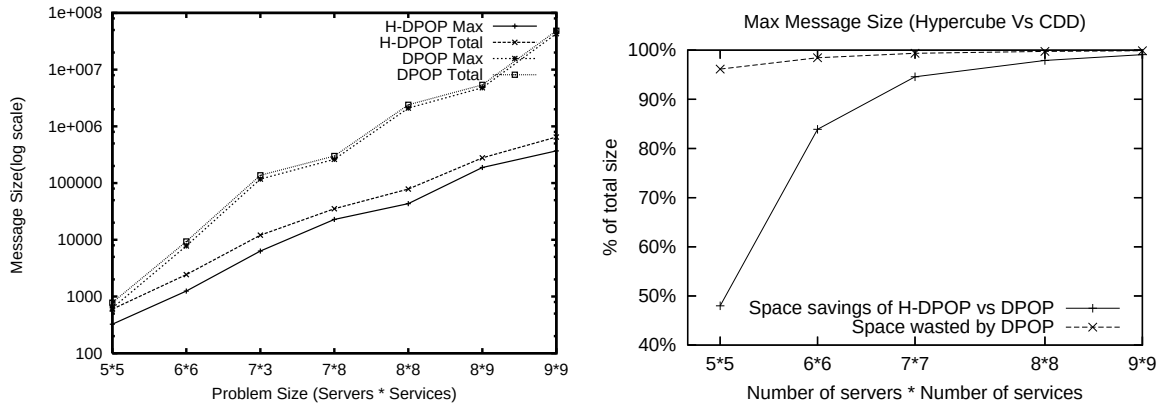
We generated random problems of different sizes, with a random number of soft constraints among variables. All-different hard constraints are introduced thus making the constraint graph fully connected. The size of the hypercube is the number of entries in the hypercube, the size of the CDD Message is the number of entries in the Util array combined with the logical size of CDDTree (each entry in the CDDNode corresponds to 1 unit in the space measurement, links to children are also counted as 1 unit).

Figure 5.5(a) shows the maximal/total message size in H-DPOP versus DPOP. Problem size is denoted by $m * n$ implying m variables each having the same domain of size n. Results show that H-DPOP is much superior to DPOP for all problem sizes, culminating with the largest problems (9 servers $\times$ 9 services) where H-DPOP produces 3 orders of magnitude smaller messages and smaller total message.

Figure 5.5(b) shows the effect of problem size on space savings provided by CDDs. We count the percentage of unfeasible assignments carried in the UTIL messages, and we plot this as "wasted space in DPOP". We see that the space wasted by DPOP is above 90%, and for larger and more difficult problems, close to 100%. In contrast, CDDs enable H-DPOP to avoid this problem. Even though CDDs introduce the overhead of representing the CDDTree explicitely, overall the space savings they provide by not sending infeasible combinations more than compensates: savings start around 48% for small problems (5*5), and increase with problem size, up to 99% for 9*9 problems.

5.4.1.2 Random Graph Coloring Problems

We performed experiments on randomly generated distributed graph coloring problems. In our setup each node in the graph is assigned an agent (or a variable in DCOP terms). The constraints among agents define the cost of having a particular color combination. The cost of two neighboring agents choosing the same color is kept very high (10000) to disallow such combinations. The *domain* of each agent is the set of available colors. The mutual task of all the agents is to find an optimal coloring



(a) Total/Maximum MessageSize

(b) Space Savings increase with problem size

Figure 5.5: Query placement problems: H-DPOP vs DPOP performance

assignment to their respective nodes.

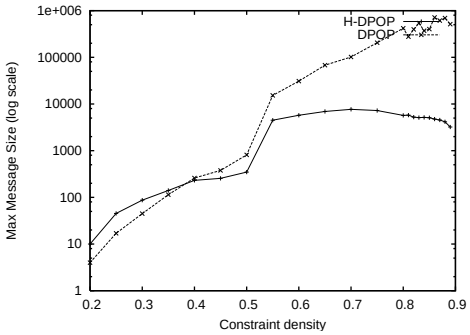
For generating these graphs we have two parameters-number of agents and the constraint density. We keep the number of agents fixed to 10. We start with a fully connected graph and remove the edges successively until we reach the desired constraint density and the problem is still connected.

Figure 5.6 shows the results on a 10 nodes randomly generated problem for a range of constraint densities (0.2-0.89). The problems within densities 0.2-0.5 were 4-colorable (implying domain size 4). The problems from 0.5-0.9 were 6-colorable. For statistically sound results for each constraint density we generated 50 random problems and the results shown are the average of 50 runs.

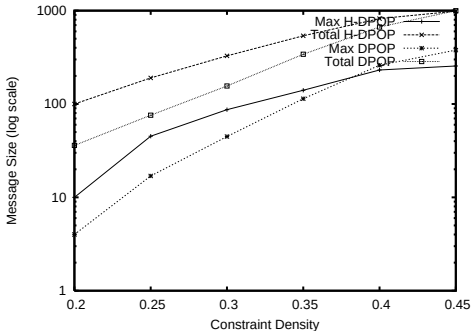
Figure 5.6(a) shows the full spectrum of performance of H-DPOP vs DPOP in terms of the maximum/total message size. For accounting the message size we take into account the number of util values in the hypercube for DPOP, for the H-DPOP we count the length of the UTIL array and the (logical) size of the CDD tree in the CDD Message. As can be seen, H-DPOP is better for most of the regions (density 0.4-0.89) except for densities from 0.2-0.4. To understand the characteristics of H-DPOP we divide the densities into three regions- low density (0.2-0.4), medium density (0.4-0.7) and high density (0.7-0.9).

For the *low density* region (Figure 5.6(b)) DPOP performs better than H-DPOP. The explanation is the same as in Section 5.4.1.1: at low density the size of the hypercube is small. CDDs at low density do not provide sufficient pruning to overcome the overhead introduced by the size of the CDDTree in the CDDMessage.

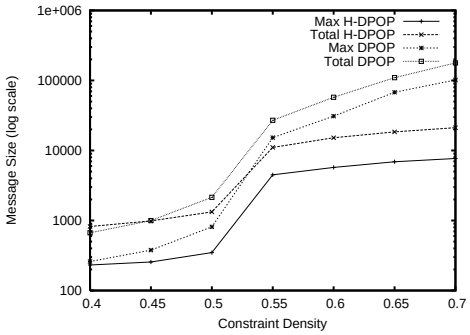
For the *medium density* region (Figure 5.6(c)) H-DPOP is much better than DPOP. The sizes of both hypercubes and CDDMessages increase with density. This is the expected behavior as with the increas-



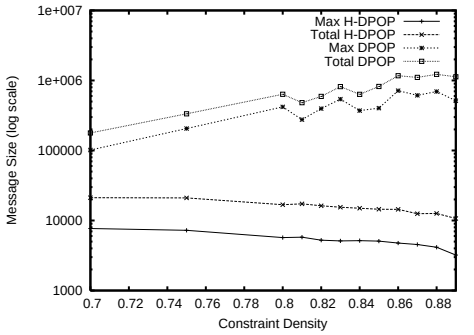
(a) Random Coloring



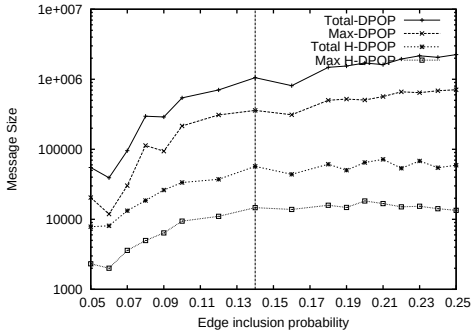
(b) Random:Low density,4 coloring



(c) Random:Medium density, 6 coloring



(d) Random: High density,6 coloring



(e) NQueen graphs

Figure 5.6: Graph Coloring: H-DPOP vs DPOP performance

ing constraint density the width increases leading to exponential increase in message size. However with CDDs we still get much space savings.

The *high density* region (Figure 5.6(d)) provides interesting results for the H-DPOP. In DPOP, as expected, the maximum message size increases with the density. However we see an opposite trend in H-DPOP, instead of CDDMessage size increasing with the density, it starts decreasing. The reason is that at high constraint density the extent of pruning done by CDDs is also very high. So although the problem becomes more complex with high connectivity, the increased pruning by the CDDs overcome this increase and at very high densities the pruning dominates the increase in problem complexity.

5.4.1.3 NQueens problems using graph coloring

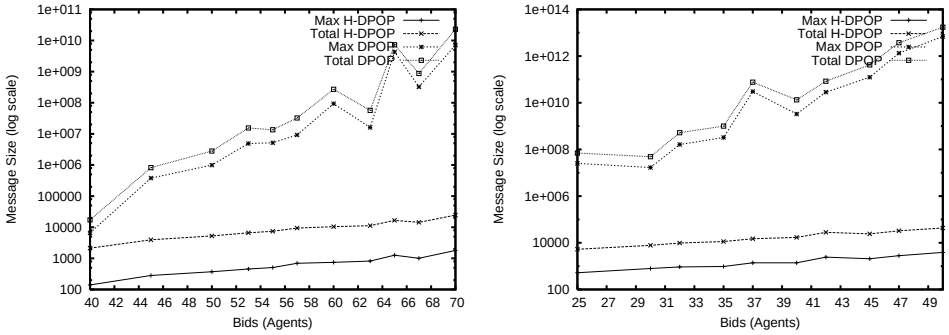
For an $n \times n$ chessboard a queen graph contains n^2 nodes, each corresponding to a square of the board. Two nodes are connected by an edge if the corresponding squares are in the same row, column, or diagonal. The intuition behind using graph coloring on a queen graph is that we can place n sets of n queens on the board so that no two queens of the same set attack each other if the chromatic number of the graph is at least n.

For our experiments we took the problems from Stanford Graphbase ([113]). For a 5-colorable 5×5 queen graph (width 19, 25 agents, density 0.53) DPOP was unable to execute (maximum message size 19073486328125). H-DPOP successfully executed in 15 seconds with a maximum message size of 9465, achieved through the high pruning power of the CDDs.

However, for board sizes 6×6 (7 colorable with width 31, density 0.46) and above H-DPOP was also unable to execute due to increased width and domain size. Relaxing a highly constrained problem is a well known technique in CSP literature. We adopt this technique into generating queen graphs so that the inclusion of any edge in the graph is done with a probability p. If this probability is 1 we get the complete queen graph.

We experimented by varying this probability from 0.05 to 0.25 for 6×6 board with the resulting graphs being 4-colorable. For each datapoint we took the average of 50 randomly generated problems. Graph remains 4-colorable until p=0.25 and increasing the p beyond increases the coloring number. A direct implication of this fact is that at p=0.25 the graph is highly constrained *with respect to* the coloring number 4. This observation lead us to believe that the nature of H-DPOP and DPOP should be similar to the random coloring experiments.

Figure 5.6(e) shows the result for maximum and total message size against the probability p. The dotted vertical line at x=0.14 divides the graph into two regions. For the first region both H-DPOP and DPOP increase in the maximum message size. However as the density (which is directly related to p) increases we see the same trend as in random problems. DPOP continues to increase in maximum size but the size in H-DPOP remains constant ($p \in$ [0.14, 0.20]), and it starts decreasing in the region



(a) Paths Distribution

(b) Arbitrary Distribution

Figure 5.7: Combinatorial Auctions: H-DPOP vs DPOP comparison

[0.20,0.25].

5.4.1.4 Winner Determination in Combinatorial Auctions

Combinatorial Auctions (CA) provide an efficient means to allocate resources to multiple agents. In CA bidders can bid on a bundle of goods in addition to single item bidding. This provides for complementarity and substitutability among the goods. In our experimental setting there is a single seller and multiple buyers (agents). The agents are distributed (geographically or logically) and have information about *only those* agents with whom their bids overlap. The mutual task of agents is to find a solution (assign winning or losing to bids) which maximizes the seller's revenue providing a feasible solution (no overlap among winning bids).

In our formulation we do search through the constraint network of bids presented (rather than considering all possible bids). Such a formulation has been shown to be very effective in CABOB[183, 185] and BOB[184]. However we do not intend to compare with these approaches as they are both centralized and use linear programming to augment the search method (not feasible in distributed setting).

The *variables* in our setting are the *bids* presented by the agents. Each agent is responsible for the bid it presents. The *domain* of each variable is the set $\{wining,\ losing\}$. Hard constraints are formulated between bids sharing one or more goods, disallowing several of them to be assigned *winning*. The value of each bid is modeled as an unary constraint on the associated variable.

We generated random problems using CATS (Combinatorial Auctions Test Suite[122]) using the *paths* and *Arbitrary* distributions. For the paths distribution the number of bids was varied for a fixed number of goods (100). Each agent is allowed to present only one bid. In *paths* distribution goods are the edges in the network of cities. Agents place bids on a path from one city to other based on their

utilities. In our setting we fixed the number of cities to 100 with initial connection 2 (link density). Since the city network structure is fixed, as the number of bids increases we expect a higher number of bids to overlap with each other and increase the problem complexity. For the *Arbitrary* distribution we use all the default CATS parameters. The number of goods is 50, and the number of bids varies from 25 to 50 increasing the complexity of the problem. Each data point is obtained as the average of 20 instances.

Figure 5.7(a) shows a comparison of DPOP with H-DPOP (average of 20 problems for each data-point) on *paths* distribution. DPOP as expected increases in message complexity with the number of bids. The pruning provided by H-DPOP is very high (around 99% of hypercubes) and increases with number of bids. Because of such high pruning H-DPOP runs on problms with very high width (35, bids=70) where memory requirements for DPOP are prohibitively expensive. We see a similar trend for the *arbitrary* distribution (figure 5.7(b)). H-DPOP is much superior to DPOP and provides very high pruning.

5.4.2 H-DPOP vs NCBB: Search Space Comparison

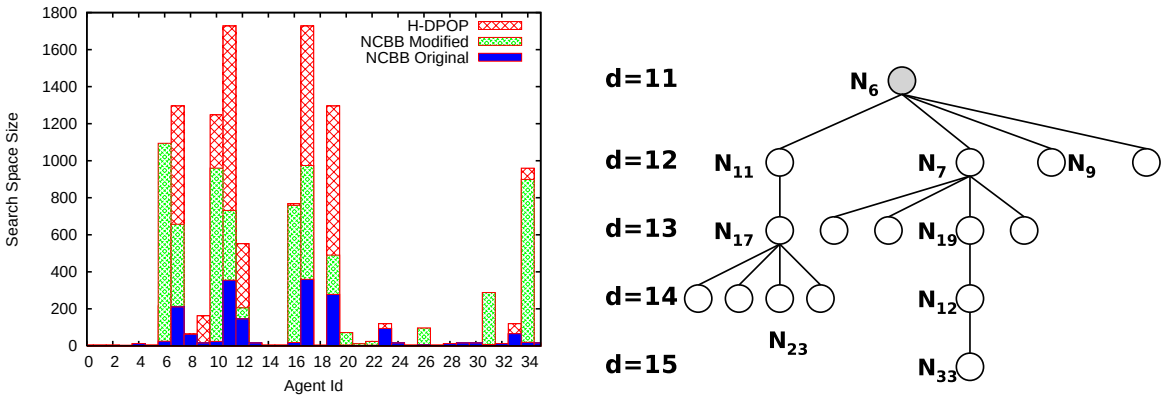
This second set of experiments (H-DPOP vs NCBB) compares the search space explored and message exchanges in H-DPOP vs different versions of NCBB. For space comparisons we compare the logical sizes of the corresponding units (hypercubes in DPOP, CDDs in H-DPOP and total cache size in search with caching).

5.4.2.1 H-DPOP vs NCBB: N-Queens

We performed a comparison of search space using the number of cache entries in NCBB's different versions and the number of util values in H-DPOP (excluding the size of CDD Tree for fair comparison) at each agent on the graph coloring problem. We selected a particular instance of queen graph (6×6 board, p=0.2, domain size=4, width=9). Our aim was to find a 4 coloring of the graph optimizing the costs assigned for color combinations. As the performance of any branch and bound search is cost dependent we generated 50 random instances of the same problem differing in the cost assignment to color combinations, each data-point is an average of 50 instances.

As stated in Section 5.3.1.1, NCBB uses much smaller cache size for all agents (Figure 5.8(a)). The reason is the non inclusion of Sep_i assignments for which subtrees do not provide a solution within the bounds. The cache size in $NCBB^*$ is similar for most of the agents to the message size in H-DPOP. There are a few cases (for agents 7,10,11,12,17,19,33) in which $NCBB^*$ is better than H-DPOP.

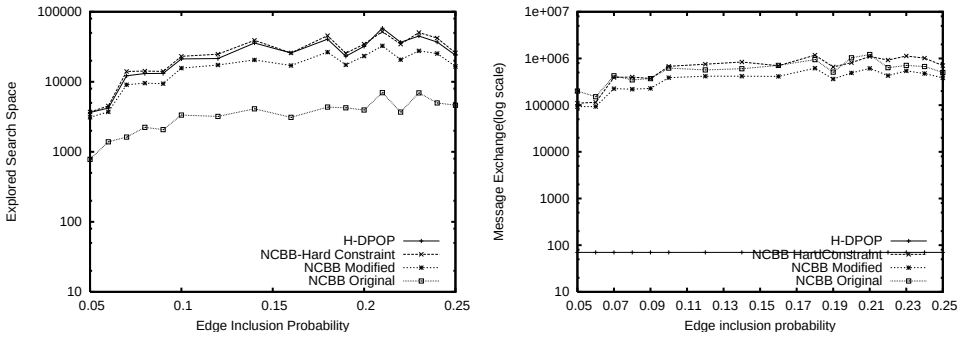
The relevant part of the DFS tree (with depth 15) for this problem is shown in figure 5.8(b), all nodes without any children are the leaves. A deeper look into the DFS arrangement suggests that all the nodes with size variations are in the lower part of the DFS tree. A search algorithm will have tighter



(a) Explored Search Space

(b) DFS tree for NQueens

Figure 5.8: NQueen Problems: H-DPOP vs NCBB Search Space comparison



(a) Explored Search Space

(b) Message Count

Figure 5.9: NQueen Problems (full range): H-DPOP vs NCBB comparison

upper bounds on the solution cost when it is expanding a high depth node, so it is natural that the effect of bounding is more pronounced for such nodes. On the contrary H-DPOP does not make use of any bounding, it prunes only the inconsistent combinations. Hence it takes more space at such nodes lower in the DFS.

An interesting result is that at the node with maximum size (Agent 6, with highest width=9) H-DPOP is much better (with size 216) as compared to cache size of 1094 in $NCBB^*$. At high width regions $NCBB^*$ does not provide good pruning (based only on bounding) however H-DPOP prunes many combinations based on consistency check. This is consistent with our previous results that at highly constrained regions H-DPOP provide very high pruning and almost negates the effect of increasing complexity.

Figure 5.9 compares NCBB and H-DPOP on the full range of 6×6 board size queen problems. The problems are same as used in Section 5.4.1.2 for solving NQueen problem using graph coloring. For each data point there are 20 randomly generated instances. As we can see from the explored search space graph (figure 5.9(a)) both $NCBB^*$ and H-DPOP explore nearly similar size search space. *NCBB Hard Constraint* explores marginally larger search space than $NCBB^*$ and as expected its search space size is very similar to H-DPOP since both H-DPOP and *NCBB Hard Constraint* do pruning based only on hard constraints.

There is not a dramatic benefit of bounding on the search between $NCBB^*$ and *NCBB Hard Constraint*. This further strengthens our claim that the major portion of pruning is attributed to the hard constraints which are exploited more efficiently by H-DPOP. One important advantage of H-DPOP is that it uses much less messages than NCBB or $NCBB^*$ (figure 5.9(b)). Even for the simpler problems (with p = 0.05), $NCBB$ uses far more number of messages than H-DPOP which always has a constant message count (70). This advantage coupled with nearly equivalent explored search space make H-DPOP much superior to a branch and bound scheme like NCBB.

5.4.2.2 H-DPOP vs NCBB: Combinatorial Auctions

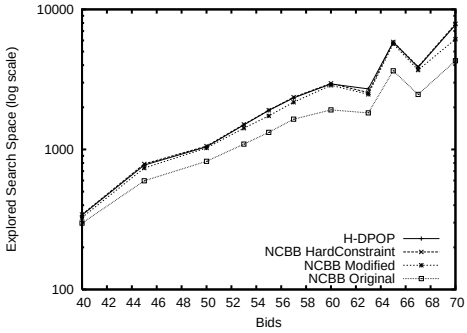
In this section we compare NCBB's different versions and H-DPOP on two metrics: explored search space and messages exchanged. The comparisons are shown in figure 5.10. The data set used is the same as in the previous section (H-DPOP vs DPOP on CA).

Notably explored search space is similar for both $NCBB^*$, *NCBB Hard Constraint* and H-DPOP for all bids. *NCBB Original* uses smaller cache size as it does not caches all combinations. The difference between *NCBB Hard Constraint* and *NCBB Modified* is again minimal suggesting that only the hard constraints play the vital role for pruning.

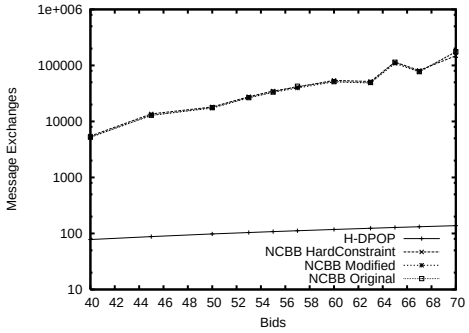
With respect to the message exchanges H-DPOP is much superior to all versions of NCBB on both paths and arbitrary distribution (figure 5.10). There is a slight difference in the number of messages between *NCBB Modified* and NCBB original but it is small to be visible on graph. *NCBB Modified* uses less number of messages (by around 5%).

Interestingly on the arbitrary problems (figure 5.10(d)) *NCBB Hard Constraint* is slightly better than its other two counterparts in terms of message exchanges. We found out that this trend occurs because *NCBB Hard Constraint* backtracks whenever it finds a single inconsistency in the partial solution. However both NCBB original and NCBB Modified tolerate inconsistent solutions until they find a better one. Figuring out the upper bound (the cost of violating one hard constraint) on the consistent solution makes NCBB and NCBB Modified to exchange extra messages.

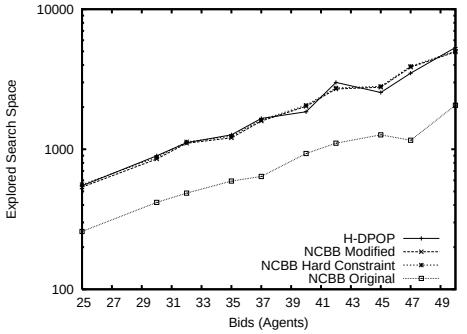
Once again in these set of experiments we have shown that explored search space is similar in both NCBB and H-DPOP, with H-DPOP requiring only a linear number of messages. Also the effect of



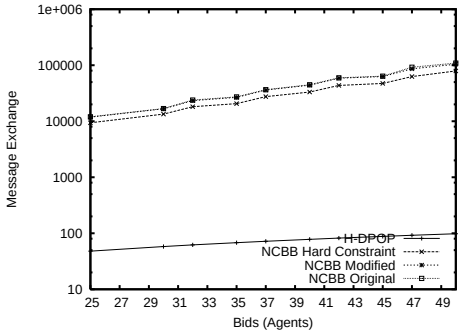
(a) Paths: Explored Search Space



(b) Paths: Message Count



(c) Random: Explored Search Space



(d) Random: Message Count

Figure 5.10: Auctions: H-DPOP vs NCBB comparison

bounding is negligible on pruning the search space as main pruning is provided by the hard constraints.

5.5 Related work

H-DPOP draws mostly from the dynamic programming algorithm DPOP (Chapter 4), and Constraint Decision Diagrams (Cheng and Yap[34]). DPOP produces large arity relations that are sent over the network. On the other hand, CDDs can take advantage of hard constraints to represent compactly such large arity relations, thus being a well suited alternative for minimizing network traffic and memory requirements for DPOP.

Recently, And/Or Multi-valued Decision Diagrams (AOMDDs) have been introduced by Mateescu and Dechter in[136]. They first arrange the problem as a pseudo-tree (of which DFS is a special case). Subsequently, on that pseudotree structure, they start a bottom-up compilation, by computing (and subsequently joining) high-arity relations (as in DPOP). However, their purpose is to have a compact

compilation of the entire constraint network in the root node. Therefore, they do not execute projections at each node along the way, thus obtaining a large AOMDD at the root, that represents the entire network. AOMDDs are space- and computation-exponential in the induced width of the DFS ordering used.

In principle, CDDs are OR-based structures, so for a complete compilation of the network, they are exponential in the path-width [1] of the problem, rather than exponential in the induced width. Therefore, they could be less space-efficient than AOMDDs. However, since each variable projects itself out of the outgoing message, our CDD representations are also guaranteed to be only exponential in the induced width of the DFS ordering used, as opposed to exponential in the problem size.

Wilson[221] introduced SLDDs (Semiring-Labelled Decision Diagrams), a generalization of CDDs to semiring structures. Our dynamic programming framework (DPOP) is easily extendable to semiring structures as well, by using SLDDs instead of CDDs as data structures for the message exchange. As CDDs, SLDDs are also OR-based structures, which means that they are size-exponential in the path-width of the problem. However, for the same reasons cited above, SLDDs applied in our context (variable elimination along a DFS tree) would also be exponential only in the induced width as opposed to the path width.

5.6 Summary

This chapter introduced H-DPOP, a new algorithm for constraint optimization based on DPOP. H-DPOP applies consistency techniques to reduce message size and memory requirements in DPOP by using CDDs. H-DPOP is an orthogonal technique, which means it can be combined with other extensions of DPOP like LS-DPOP, MB-DPOP, A-DPOP, etc. Experimental results show that in cases where the problems are highly constrained, this representation allows for as much as 99% space savings as compared to the basic dynamic programming approach.

The second contribution of this chapter is an extensive comparison with search algorithms, which compares the pruning achieved by search with the one achieved by using CDDs in dynamic programming. Pruning techniques are very natural to search algorithms, and can boost their performance significantly. Introducing CDDs into DPOP gives dynamic programming algorithms similar pruning capabilities, and yields similar performance improvements. Our extensive analysis shows that although pruning in H-DPOP works bottom-up as opposed to top-down in search, similar effects are obtained, and the portions of the search space explored by H-DPOP and search are very similar.

There are many realistic scenarios where hard constraints restrict the search space significantly. For example, several types of auctions have this property: auctions where agents bid on paths in space like railroad auctions, auctions for airport time slots, etc. Other examples include advanced versions of

[1] Path-width is the induced width of linear orderings

the service allocation problem, or scheduling with resource constraints. All these problems are large, highly constrained problems, and can be efficiently solved by H-DPOP.

We conclude that in many applications such as those described above, H-DPOP is an excellent approach, because it combines the best of both search and dynamic programming: it requires only a linear number of messages like dynamic programming (i.e. low networking overhead), and by using CDDs and their pruning power we can effectively limit the size of these messages, like in search.

Part III

Tradeoffs

In this part of the thesis we discuss tradeoffs in DCOP along 3 dimensions: solution quality (complete vs. incomplete algorithms), memory requirements (linear / polynomial / exponential), communication requirements (few large messages vs. many small messages), degree of distribution (fully distributed algorithms vs. partial centralization algorithms).

Chapter 6

Tradeoffs between Memory/Message Size and Number of Messages

In this chapter we discuss possible tradeoffs that variants of the DPOP algorithm can offer for problems with high induced width, where the basic DPOP algorithm cannot be applied due to memory or communication restrictions. The chapter is organized as follows: we start with a quick recapitulation of DPOP, and its main features in Section 6.1. Then we present the first contribution of this chapter: a generic, configurable framework for identifying and isolating difficult subproblems *of high width, which cannot be solved with the high-performance DPOP propagations. A distributed algorithm to this effect is presented in Section 6.2. Once such difficult subproblems are identified, they can be solved with any of a number of alternative methods, and the partial results integrated in the overall DPOP propagation.*

The second contribution is MB-DPOP, a configurable algorithm that uses the cycle-cutset *idea to offer a tradeoff between the amount of memory used and the number of messages. MB-DPOP is shown to perform up to 5 orders of magnitude better than ADOPT, the state of the art in memory-bounded search.*

The third contribution is O-DPOP, a hybrid of best-first search and dynamic programming, which combines some advantages of both worlds: First, it uses messages whose size only grows linearly (as in search) with the treewidth of the problem. Second, by letting agents explore values in a best-first order, it avoids incurring always the worst case complexity as DPOP, and on average it saves a significant amount of computation and information exchange.

6.1 DPOP: a quick recap

The basic dynamic programming algorithm *DPOP* has been introduced in Chapter 4. *DPOP* is an instance of the general bucket elimination scheme from[51], which is adapted for the distributed case, and uses a DFS traversal of the problem graph as an ordering. *DPOP* has 3 phases:

1. **DFS traversal:** a DFS traversal of the graph is done using a distributed DFS algorithm, like in[160], which works for any graph requiring a linear number of messages. The outcome is that all nodes consistently label each other as parent/child or pseudoparent/pseudochild, and edges are identified as tree/back edges. The DFS tree serves as a communication structure for the other 2 phases of the algorithm: UTIL messages (phase 2) travel bottom-up, and VALUE messages (phase 3) travel top down, only via tree-edges.

2. **UTIL propagation**: the agents (starting from the leaves) send *UTIL* messages to their parents. The subtree of a node X_i can influence the rest of the problem only through X_i's separator, Sep_i. Therefore, a message contains the optimal utility obtained in the subtree for each instantiation of Sep_i. Thus, messages are size-exponential in the separator size (which is in turn bounded by the induced width).

3. **VALUE propagation:** a top-down optimal assignment propagation phase is initiated by the root, when phase 2 has finished. Each node determines its optimal value based on the computation from phase 2 and the *VALUE* message it has received from its parent. Then, it sends this value to its children through *VALUE* messages.

DPOP complexity:

- number of messages: linear in the number of agents
- message size: largest UTIL message is space-exponential in the width of the DFS ordering used.

6.2 DFS-based method to detect subproblems of high width

We have seen that DPOP's memory requirements are exponential in the *induced width* of the constraint graph, which may be prohibitive for problems with large width. For such cases, we introduce the control parameter k which specifies the maximal amount of inference (maximal message dimensionality). This parameter is chosen s.t. the available memory at each node is greater than d^k, (d is the domain size).

We propose in this section an algorithm that identifies subgraphs of the problem (clusters) that have width higher than k, where due to memory limitations, it is not possible to perform full inference as in DPOP. Nodes *inside* such clusters will have to recourse to some other techniques (see Section 6.3, Section 8, Section 7.1, Section 7.2). Nodes *outside* these clusters can perform the normal DPOP UTIL and VALUE propagations, which have the advantages we previously discussed (optimality guarantees, low overhead, etc). The result is that in most parts of the problem, high-performance DPOP propagations are used, and only in minimal, high-width subproblems we have to recourse to other alternatives.

Definition 19 (Cluster node) *Given a DFS tree and a number k, a node X_i in the DFS is called a* cluster-node *iff $|Sep_i| > k$.*

A cluster is bounded at the top by the lowest node in the tree that has separator of size k or less. We call these top-bounding nodes *cluster roots* (CR).

Definition 20 (Cluster root node) *Given a DFS tree and a number k, a node X_i in the DFS is called a* cluster-root node *iff $\exists X_j \in C_i$ s.t. $|Sep_j| > k$, and $|Sep_i| \leq k$.*

Definition 21 (Cluster of width greater than k) *Given a DFS tree and a number k, a cluster C_r of width greater than k is a set of nodes which are all labeled as cluster node or cluster root, and there is a tree path between any pair of nodes $X_i, X_j \in C_r$, that goes only through cluster nodes.*

Briefly, the clusters are identified in a bottom-to-top pass on the DFS tree. The process works by labeling the nodes with separator size larger than k as *cluster-nodes*, and including them in a cluster. Subsequently, inside a cluster, we use an alternative UTIL propagation which uses less memory than the normal DPOP propagations. The goal is to find an optimal solution for each cluster for each assignment of the variables in the separator of the cluster root. The results are cached ([8, 42, 132]) by the respective cluster roots and then integrated as normal UTIL messages into the overall DPOP-type UTIL propagation. Subsequently, during the final VALUE propagation phase, the results cached in the UTIL phase are retrieved, and the VALUE propagation continues as in normal DPOP.

If w is the induced width of the problem given by the chosen DFS ordering, depending on the value chosen for k, we have 3 cases:

Algorithm 9 *LABEL-DFS - a protocol to determine the areas of high width.*

LABEL-DFS($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$, k) (assumes a DFS tree created with Algorithm 3). Each agent X_i does:

Labeling protocol:

wait for all $LABEL_{j \to i}$ msgs from children
$Sep_i = \cup_{X_j \in C_i} Sep_j \cup P_i \cup PP_i \setminus X_i$
if $|Sep_i| > k$ **then** label self as *cluster-node*
else
  if $\exists X_j \in C_i$ *such that* $|Sep_j| > k$ **then** label self as *cluster-root*
  else label self as normal
send $LABEL_i^{P_i} = [Sep_i]$ to P_i

1. If $k = 1$, only linear messages are used, and memory requirements are also linear.
2. If $k < w$, full inference can be performed in areas of width lower than k, and an alternative processing in areas of width higher than k. Memory requirements are $O(exp(k))$.
3. If $k \geq w$, full inference is done throughout the problem, and the algorithm is equivalent with DPOP (i.e. full inference everywhere). Memory requirements are $O(exp(w))$.

Intuitively, the larger the k, the less need for identifying clusters, and the larger the parts of the problem where standard DPOP is applied.

In the next sections, we will discuss a number of extensions of DPOP, which all identify complex subproblems in this way, and then apply different techniques to deal with them: Section 6.3 discusses MB-DPOP, which applies cycle-cutsets to reduce message size, at the expense of an increase in the number of messages. Section 8 introduces the PC-DPOP algorithm, which allows for the partial centralization of difficult subproblems. Section 7.1 introduces the LS-DPOP algorithm, which applies local search in difficult subproblems, and limited dynamic programming to guide it. Section 7.2 introduces the A-DPOP algorithm, an approximation scheme which limits the size of the messages to $O(d^k)$ and propagates upper and lower bound messages in subproblems with high width.

In the following Section 6.2.1, we explain how to determine high-width areas using Algorithm 9.

6.2.1 DFS-based Label propagation to determine complex subgraphs

This is an intermediate phase between the DFS and UTIL phases, and it has the goal to delimit high-width clusters. We emphasize that this process is described as a separate phase only for the sake of clarity; these results can be derived with small modifications from either the original DFS construction algorithm, or the subsequent UTIL phase.

Labeling works bottom-up like the UTIL phase. A $LABEL_{i \to P_i}$ message is composed of the list of nodes in the *separator* Sep_i of the sending node X_i. Each node X_i waits for $LABEL_{j \to i}$ messages

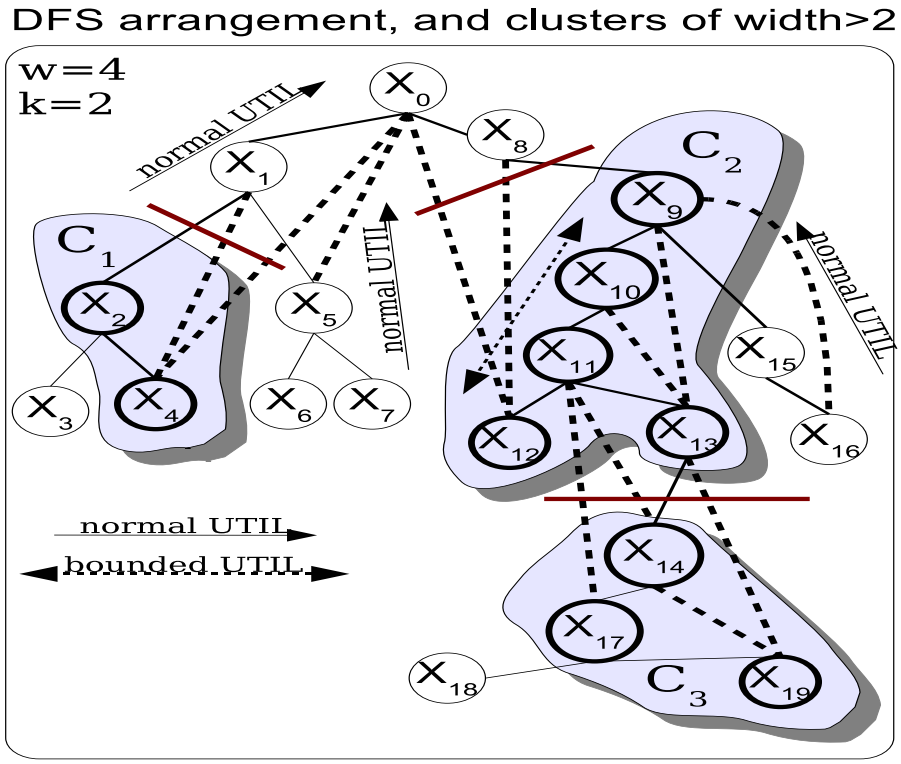


Figure 6.1: *A DFS tree of width $w = 4$. Minimal areas of high width are identified based on the node separator size (shaded clusters C_1, C_2 and C_3). In low-width areas the normal UTIL propagation is performed. In high width clusters, alternative UTIL propagations are used, and cluster roots (X_2, X_9, X_{14}) cache intermediate results.*

from its children $X_j \in C_i$, computes its own label $LABEL_{i \rightarrow P_i}$, and sends it to its parent P_i. The process finishes when the root has received $LABEL$ messages from all its children.

Recall that each node X_i can easily determine its separator recursively, as in Equation 3.2. If the separator Sep_i of X_i contains more than k nodes, this means that the UTIL message that normal DPOP would send would exceed the size limit $O(exp(k)$. Therefore, X_i is part of a high-width cluster, and labels itself as a *cluster-node*. If a node X_i has separator size equal to k or less, then the node could be in one of these two cases:

- if X_i has any child which is a cluster-node (i.e. the separator of the child is larger than k), then X_i is a cluster-root
- if X_i has only children with separators equal to k or smaller than k, then X_i is a normal node

Example 11 *in Fig. 6.1, let $k = 2$. Light nodes (e.g. X_0, X_1, X_3, etc.) all have separator size less than 2. Bold nodes on the other hand have separator size greater than 2 (e.g. node X_{12} has $Sep_{12} = \{X_0, X_8, X_{11}\}$). The shaded areas are the clusters C_1, C_2 and C_3 identified after running*

Algorithm 9.

6.3 MB-DPOP(k): Trading off Memory vs. Number of Messages

This section introduces MB-DPOP(k) (Algorithm 10), a new hybrid algorithm that can operate with bounded memory. MB-DPOP(k) is controlled by a parameter k which allows the user to specify the maximal amount of inference (maximal message dimensionality). This parameter is chosen such that the available memory at each node is greater than d^k, (d is the domain size).

MB-DPOP(k) operates in the framework of Section 6.2 for detecting high-width clusters, where it is not possible to perform full inference as in DPOP. Clusters of high width are explored with bounded propagations using the idea of *cycle-cuts*[51]. The cycle-cut nodes (CC) are a subset of nodes such that once removed, the remaining problem has width k or less. Subsequently, in each cluster all combinations of values of the CC nodes are explored using sequential k-bounded UTIL propagations. Therefore, in these areas of high width, MB-DPOP offers a tradeoff of the linear number of messages of DPOP for polynomial memory. In areas of low width, MB-DPOP uses the normal, high performance DPOP propagations.

The overall behavior of MB-DPOP(k) is as follows: if w is the induced width of the problem given by the chosen DFS ordering, depending on the value chosen for k, we have 3 cases:

1. If $k = 1$, only linear messages are used, and a full cycle cutset is determined. MB-DPOP(1) is similar to the AND/OR cycle cutset scheme from[135]. Memory requirements are linear.

2. If $k < w$, MB-DPOP(k) performs full inference in areas of width lower than k, and bounded inference in areas of width higher than k. Memory requirements are $O(exp(k))$.

3. If $k \geq w$, full inference is done throughout the problem; MB-DPOP(k) is then equivalent with DPOP (i.e. full inference everywhere). Memory requirements are $O(exp(w))$.

Partial results within each cluster are cached ([8, 42, 132]) by the respective cluster root and then integrated as messages into the overall DPOP-type propagation. This helps reduce the overall complexity from exponential in the total number of cycle-cut nodes to exponential in the largest number of cycle cuts in a single cluster.

The rest of this section is organized as follows: we explain how to determine high-width areas and the respective cycle-cuts (Section 6.3.1) and what changes we make to the *UTIL* and *VALUE* phases (Section 6.3.2 and Section 6.3.3). The complexity of the algorithm is analyzed formally in Section 6.3.4. In Section 6.3.5, we compare MB-DPOP with ADOPT[141], the current state of the art in distributed search with bounded memory. MB-DPOP consistently outperforms ADOPT on 3 problem domains, with respect to 3 metrics, providing speedups of up to 5 orders of magnitude.

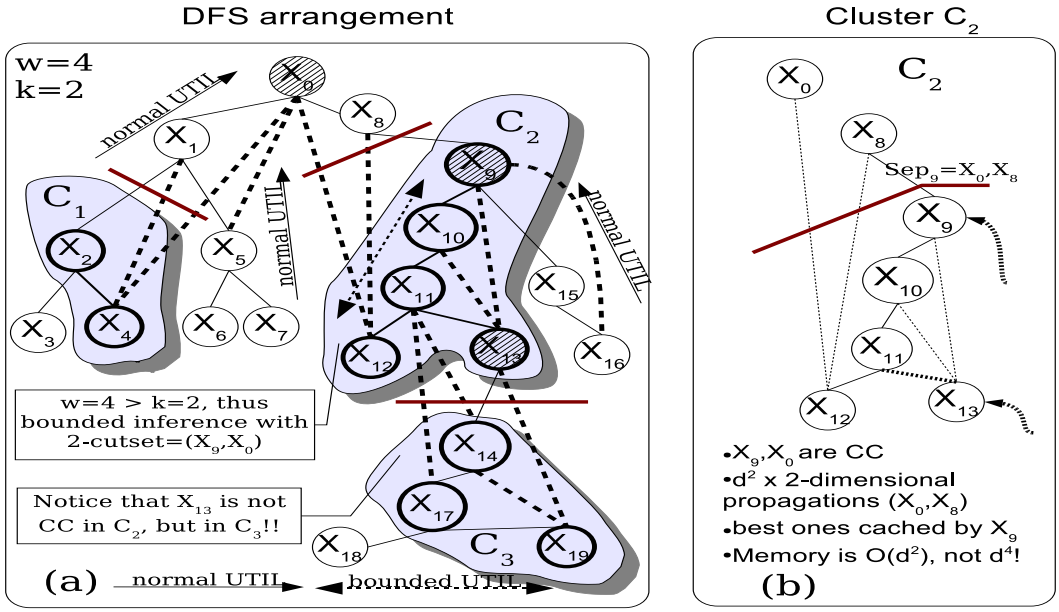


Figure 6.2: *A DFS tree of width w=4. In low-width areas the normal UTIL propagation is performed. In high width areas (shaded clusters C_1, C_2 and C_3 in (a)) bounded UTIL propagation is used. All messages are of size at most d^k. Cycle-cut nodes are hashed (X_0, X_9, X_{13}), and X_2, X_9, X_{14} are cluster roots. In (b) we show a 2-bounded propagation.*

6.3.1 MB-DPOP - Labeling Phase to determine the Cycle Cuts

This is an extension of the framework of Section 6.2 for detecting high-width subproblems, where it is not possible to perform full inference as in DPOP. In addition to grouping nodes into clusters of high width, this extension also designates a subset of these nodes to be cycle-cut nodes (called a *w-cutset* in[23]).

As in Section 6.2, labeling works bottom-up like the UTIL phase. Each node X_i waits for $LABEL_j^i$ messages from its children X_j, computes its own label $LABEL_i^{P_i}$, and sends it to its parent P_i. In Section 6.2, label messages contain the separator of the sending node. Here, we extend them by adding to each message a list CC_i of nodes to be designated as cycle cuts. The semantics of the list CC_i sent from X_i to P_i is as follows: $\forall X_c \in CC_i$, there is a node X_j in the cluster which contains X_i, such that X_j has $|Sep_j| > k$, and X_j therefore declared X_c as a CC node. Each node computes this list through a heuristic function based on the separator of the node, and on the lists of cycle-cuts received from the children (see next section).

As the labeling process proceeds, the list of CC nodes will "accumulate" to the cluster root, which is able to send its UTIL message as in normal DPOP, since its size limit is observed. Consequently, the cluster root will send an empty CClist to its parent, as the nodes in its own cluster need not be treated as CC nodes upstream.

Algorithm 10 *MB-DPOP - memory bounded DPOP.*

MB-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$, k): each agent X_i does:

Labeling protocol:

wait for all $LABEL_j^i$ msgs from children
if $|Sep_i| \leq k$ **then**
  if $\cup CClists \neq \emptyset$ **then** label self as CR
  else label self as normal
  $CC_i \leftarrow \emptyset$
else
  let $N = Sep_i \setminus \cup CClists$
  select a set CC_{new} of $|N| - k$ nodes from N
  return $CC_i = CC_{new} \cup CClists$
send $LABEL_i^{P_i} = [Sep_i, CC_i]$ to P_i

***UTIL* propagation protocol**

wait for $UTIL_k^i$ messages from all children $X_k \in C(i)$
if $X_i =$ *normal node* **then** do UTIL / VALUE as DPOP
else
  do propagations for all instantiation of $CClists$
  if X_i *is cluster root* **then**
    update UTIL and CACHE for each propagation
    when propagations finish, send UTIL to parent

***VALUE* propagation**(X_i receives Sep_i^* from P_i)

if X_i *is cluster root* **then**
  find in cache the CC^* corresponding to Sep_i^*
  assign self according to cached value
  send CC^* to nodes in CC via VALUE messages
else
  perform last UTIL with CC nodes assigned to CC^*
  assign self accordingly
Send *VALUE*$(X_i \leftarrow v_i^*)$ to all $C(i)$ and $PC(i)$

6.3.1.1 Heuristic labeling of nodes as CC

Let $label(Sep_i, CClists, k)$ be a heuristic function that takes as input the separator of a node, the lists of cycle-cuts received from the children, and an integer k, and it returns another list of cycle cutset nodes.

It builds the set $N_i = Sep_i \setminus \{\cup CClists\}$: these are nodes in X_i's separator that are not marked as CC nodes by X_i's children. If $|N_i| > k$ (too many nodes not marked as CC), then it uses any mechanism to select from N_i a set CC_{new} of $|N_i| - k$ nodes, that will be labeled as CC nodes. The

function returns the set of nodes $CC_i = \cup CClists \cup CC_{new}$.

If the separator Sep_i of X_i contains more than k nodes, then this ensures that enough of them will be labeled as cycle-cuts, either by the children of X_i or by X_i itself. If $|Sep_i| \leq k$, the function simply returns an empty list.

Mechanism 1: highest nodes as CC The nodes in N_i are sorted according to their tree-depth (known from the DFS phase). Then, the highest $|N_i| - k$ nodes are marked as CC.

Example 12 *in Fig. 6.2, let $k = 2$. Then, $Sep_{12} = \{X_0, X_8, X_{11}\}$, $CClists_{12} = \emptyset \Rightarrow N_{12} = Sep_{12} \Rightarrow CC_{12} = \{X_0\}$ (X_0 is the highest among X_0, X_8, X_{11})*

Mechanism 2: lowest nodes as CC This is the inverse of Mechanism 1: the lowest $|N_i| - k$ nodes are marked as CC.

Example 13 *in Fig. 6.2, let $k = 2$. Then, $Sep_{12} = \{X_0, X_8, X_{11}\}$, $CClists_{12} = \emptyset \Rightarrow N_{12} = Sep_{12} \Rightarrow CC_{12} = \{X_{11}\}$ (X_{11} is the lowest among X_0, X_8, X_{11})*

6.3.2 MB-DPOP - *UTIL* Phase

The labeling phase (Section 6.3.1) has determined the areas where the width is higher than k, and the corresponding CC nodes. We describe in the following how to perform bounded-memory exploration in these areas; anywhere else, the original UTIL propagation from DPOP applies.

Let X_i be the root of a cluster. Just like in DPOP, X_i creates a $UTIL_i^{P_i}$ table that stores the best utilities its subtree can achieve for each combination of values of the variables in Sep_i. X_i's children X_j that have separators smaller than k ($|Sep_j| \leq k$) send X_i normal $UTIL_j^i$ messages, as in DPOP; X_i waits for these messages, and stores them.

For the children X_j that have a larger separator ($|Sep_j| > k$), X_i creates a $Cache$ table with one entry $Cache(sep_i)$ that corresponds to each particular instantiation of the separator, $sep_i \in \langle Sep_i \rangle$; the size of the $Cache$ table is thus exactly the same as the outgoing UTIL message, i.e. $O(exp(|Sep_i|))$.

X_i then starts exploring through k-bounded propagation all its subtrees that have sent non-empty $CClists$. It does this by cycling through all instantiations of the CC variables in the cluster. Each one is sent down to its children via a *context* message. Context messages propagate top-down to all the nodes in the cluster.

The leaves of the cluster then start a bounded propagation, with the CC nodes instantiated to the values specified in the context message. These propagation are guaranteed to involve k dimensions or

less, and they proceed as in normal DPOP, until they reach X_i, the root of the cluster. X_i then updates the best utility values found so far for each $sep_i \in \langle Sep_i \rangle$, and also updates the cache table with the current instantiation of the CC nodes in case a better utility was found.

When all the instantiations are explored, X_i simply sends to its parent the updated $UTIL_i^{P_i}$ table that now contains the best utilities of X_i's subtree for all instantiations of variables in Sep_i, exactly as in DPOP. P_i then continues the UTIL propagation as in normal DPOP, and all the complexity of the cycle cutset processing performed below in the cluster rooted at X_i is transparent to it.

Example 14 *In Figure 6.2, let $k = 2$; then $C_2 = \{X_9, X_{10}, X_{11}, X_{12}, X_{13}\}$ is an area of width higher than 2. X_9 is the root of C_2, as the first node (lowest in the tree) that has $Sep_i \leq k$. Using the Mechanism 1 for selecting CC nodes, we have X_9, X_0 as CC in C_2. X_9 cycles through all the instantiations $\langle X_9, X_0 \rangle$, and sends its child X_{10} context messages of the form $\langle X_9 = a, X_0 = b \rangle$ (only to X_{10} because X_{15} requires no cycle cutset processing, and has already sent its UTIL message to X_9). These context messages travel to all nodes in cluster C_2: X_{10}, X_{11}, X_{12} and X_{13}. Upon receiving a context message, X_{12} and X_{13} start 2-bounded UTIL propagation (X_{12} with X_{11} and X_8 as dimensions, and X_{13} with X_{11} and X_{10} as dimensions).*

6.3.3 MB-DPOP - *VALUE* Phase

The labeling phase has determined the areas where bounded inference must be applied due to excessive width. We will describe in the following the processing to be done in these areas; outside of these, the original VALUE propagation from DPOP applies.

The VALUE message that the root X_i of a cluster receives from its parent contains the optimal assignment of all the variables in the separator Sep_i of X_i (and its cluster). X_i retrieves from its cache table the optimal assignment corresponding to this particular instantiation of the separator. This assignment contains its own value, and the values of all the CC nodes in the cluster. X_i informs all the CC nodes in the cluster what their optimal values are (via *VALUE* messages).

As the non-CC nodes in the cluster could not have cached their optimal values for all instantiations of the CC nodes, it follows that a final *UTIL* propagation is required in order to re-derive the utilities that correspond to the particular instantiation of the CC nodes that was determined to be optimal. However, this is not an expensive process, since it is a single propagation, with dimensionality bounded by k (the CC nodes are instantiated now). Thus, it requires only a linear number of messages that are at most $exp(k)$ in size.

Subsequently, outside the clusters, the *VALUE* propagation proceeds as in DPOP.

6.3.4 MB-DPOP(k) - Complexity

Assume we have chosen a given k. In low-width areas of the problem, MB-DPOP behaves exactly as DPOP: it generates a linear number of messages that are at most d^k in size. Clusters are formed where the width exceeds k. Let T be such a cluster; we denote by $|T|$ the number of nodes in the cluster T, and by $|CC(T)|$ the number of cycle cut nodes in cluster T. Let T^* be the cluster such that $T^* = argmax_T|CC(T)|$ (the cluster with the largest number of cycle cut nodes). Then we have the following:

Theorem 2 (MB-DPOP Complexity) *MB-DPOP(k) requires at most $O(exp(k))$ memory at each node. MB-DPOP(k) requires at most $O(exp(|CC(T^*)|))$ messages, each of size at most $O(exp(k))$.*

PROOF. For the first part of the claim: during the initial labeling phase, each node determines the size of its separator. Nodes with separator size smaller than k act as in DPOP, and thus send messages smaller than $O(exp(k))$, and require memory smaller than $O(exp(k))$. Nodes with separator size greater than k turn to the bounded inference process, which limit the size of their messages to $O(exp(k))$.

For the second part of the claim: MB-DPOP(k) executes $d^{|CC(T)|}$ k-bounded propagation in each cluster T. Each propagation requires $|T| - 1$ messages, as each execution is similar to a limited DPOP execution. The size of these messages is bounded by d^k by construction. It is easy to see that the overall time/message complexity is given by the most difficult cluster, T^*: $O(exp(|CC(T^*)|))$ where T^* is the cluster that has the maximal number of CC nodes. □

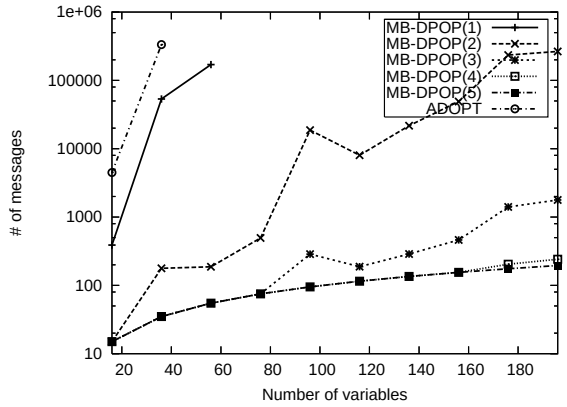
6.3.5 MB-DPOP: experimental evaluation

We performed experiments on 3 different problem domains: distributed sensor networks (DSN), graph coloring (GC), and meeting scheduling (MS). All experiments are run on a P4 machine with 1GB RAM, using the FRODO[154] simulation platform.

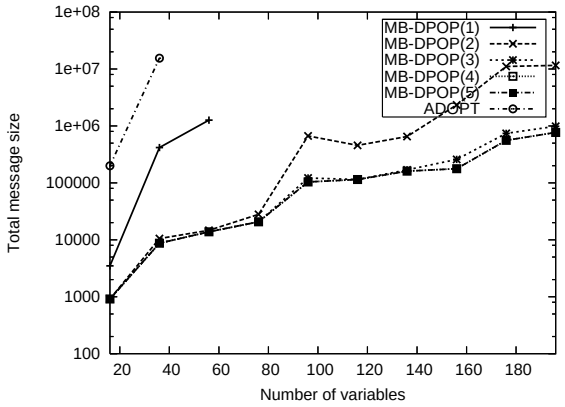
6.3.5.1 Meeting scheduling

We generated a set of relatively large distributed meeting scheduling problems. The model is as in[127], and described in detail in Section 2.3.1. Briefly, an optimal schedule has to be found for a set of meetings between a set of agents. The test instances contained from 10 to 100 agents, and 5 to 60 meetings, yielding large problems with 16 to 196 variables. The larger problems were also denser, therefore even more difficult (induced width from 2 to 5).

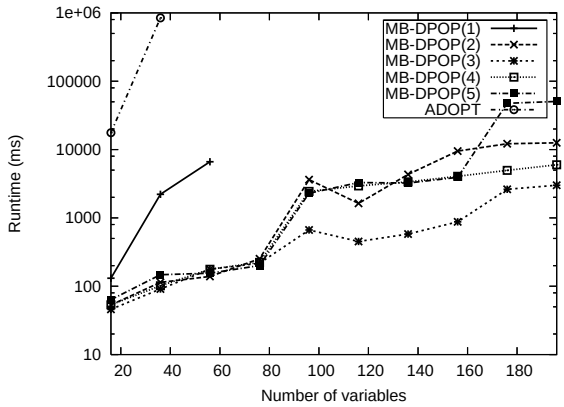
The experimental results are presented in Figure 6.3. Figure 6.3(a) shows the number of messages exchanged, and Figure 6.3(b) shows the sum of all message sizes, in bytes. Figure 6.3(c) shows the



(a) Number of messages



(b) Total size of messages



(c) Runtime in miliseconds

Figure 6.3: MB-DPOP(k) vs ADOPT - evaluation on meeting scheduling problems.

runtime in milliseconds. [1]. Please notice the logarithmic scale! ADOPT did not scale on these problems, and we had to cut its execution after a threshold of 2 hours or 5 million messages, whichever occured first. The largest problems that ADOPT could solve had 20 agents (36 variables).

We also executed MB-DPOP with increasing bounds k. As expected, the larger the bound k, the less nodes will be designated as CC, and the fewer messages will be required[2]. However, message size and memory requirements increase.

It is interesting to note that even MB-DPOP(1) (which uses linear-size messages, just like ADOPT) performs much better than ADOPT: it can solve larger problems, with a smaller number of messages. For example, for the largest problems ADOPT could solve, MB-DPOP(1) produced improvements

[1] Each data point is an average over 10 instances

[2] Mechanism 1 for CC selection was used.

of 3 orders of magnitude. MB-DPOP(2) improved over ADOPT on some instances for 5 orders of magnitude.

Also, notice that even though MB-DPOP($k > 1$) sends larger messages than ADOPT, overall, it exchanges much less information (Fig 6.3(b)). We believe there are 2 reasons for this: ADOPT sends many more messages, and because of its asynchrony, it has to attach the full context to all of them (which produces extreme overheads).

6.3.5.2 Graph Coloring

The GC problems are the same as the ones used in[127], and are available online at[151]. These are small instances (9 to 12 variables), but they are more tightly connected, and are quite challenging for ADOPT. ADOPT terminated on all of them, but required up to 1 hour computation time, and 4.8 million messages for a problem with 12 variables. The results are shown in Figure 6.4.

6.3.5.3 Distributed Sensor Networks

The DSN problems are also the same as the ones used in[127], and available online at[151]. The DSN instances are very sparse, and the induced width is 2, so MB-DPOP($k \geq 2$) always runs with a linear number of messages (from 100 to 200 messages) of size at most 25. Runtime varies from 52 ms to 2700 ms. In contrast, ADOPT sends anywhere from 6000 to 40.000 messages, and requires from 6.5 sec to 108 sec to solve the problems. Overall, these problems were very easy for MB-DPOP, and we have experienced around 2 orders of magnitude improvements in terms of CPU time and number of messages.

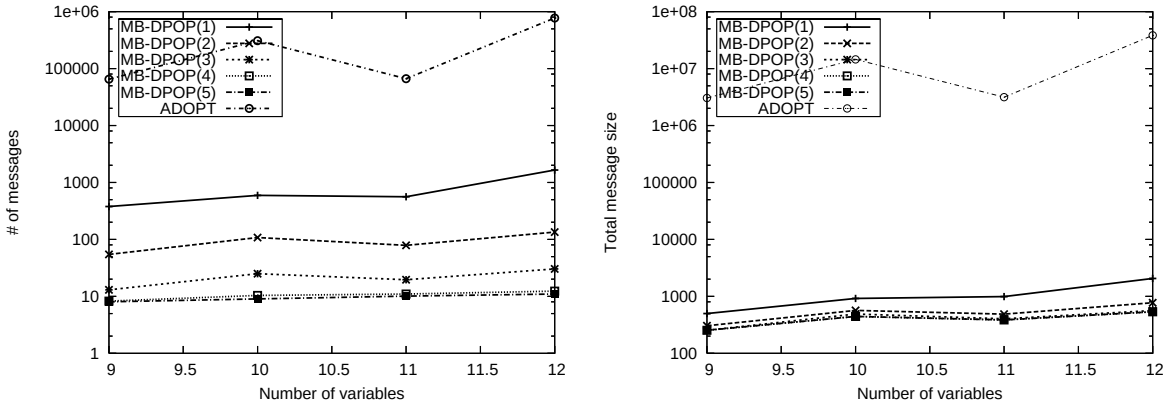
All three domains showed strong performance improvements of MB-DPOP over the previous state of the art algorithm, ADOPT. On these problems, we noticed up to 5 orders of magnitude less computation time, number of messages, and overall communication.

6.3.6 Related Work

The w-cutset idea was introduced in[177]. A *w-cutset* is a set CC of nodes that once removed, leave a problem of induced width *w* or less. One can perform search on the w-cutset, and exact inference on the rest of the nodes. The scheme is thus time exponential in $d^{|CC|}$ and space exponential in k.

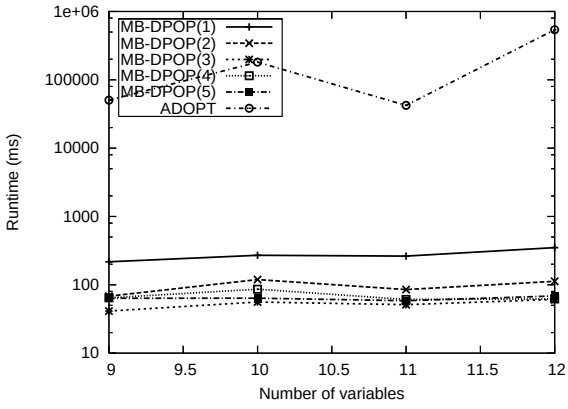
If separators smaller than k exist, MB-DPOP(k) isolates the cutset nodes into different clusters, and thus it is time exponential in $|CC(T_{max})|$ as opposed to exponential in $|CC|$. Since $|CC(T_{max})| \leq |CC|$, MB-DPOP(w) can produce exponential speedups over the w-cutset scheme.

AND/OR w-cutset is an extension of the w-cutset idea, introduced in[135]. The w-cutset nodes



(a) Number of messages

(b) Total size of messages



(c) Runtime in miliseconds

Figure 6.4: MB-DPOP(k) vs ADOPT - evaluation on graph coloring problems.

are identified and then arranged as a *start-pseudotree*. The lower parts of the pseudotree are areas of width bounded by w. Then AND/OR search is performed on the w-cutset nodes, and inference on the lower parts of bounded width. The algorithm is time exponential in the depth of the start pseudotree, and space exponential in w.

It is unclear how to apply their technique to a distributed setting, particularly as far as the identification of the w-cutset nodes and their arrangement as a start pseudotree are concerned. MB-DPOP solves this problem elegantly, by using the DFS tree to easily delimit clusters and identify w-cutsets. Furthermore, the identified w-cutsets are already placed in a DFS structure.

That aside, when operating on the same DFS tree, MB-DPOP is superior to the AND/OR w-cutset scheme without caching on the start pseudotree. The reason is that MB-DPOP can exploit situations where cutset nodes along the same branch can be grouped into different clusters. Thus MB-DPOP's complexity is exponential in the largest number of CC nodes in a single cluster, whereas AND/OR w-cutset is exponential in the total number of CC nodes along that branch. MB-DPOP has the same asymptotic complexity as the AND/OR w-cutset with w-bounded caching.

Petcu and Faltings present in [156] a distributed cycle cutset optimization method. The idea of isolating independent cyclic subgraphs appears there, too, but unfortunately there is no efficient method presented for identifying cycle cutset nodes, nor for isolating independent cyclic subgraphs. Here, the DFS traversal of the graph is an excellent way to achieve both goals. There, the separator sizes are always forced to 1, resulting in less opportunities for finding small clusters, that have a small number of cycle cuts. The inference is also bounded to $k = 1$, not allowing the algorithm to take advantage of additional memory that may be available. The complicated synchronization problems between cycles from that method are solved here by simply making each cluster root wait for complete exploration of all its cluster(s) before sending its message to its parent.

Finally, tree clustering methods (e.g. [107]) have been proposed for time-space tradeoffs. MB-DPOP uses the concept loosely, only in high-width parts of the problem. For a given DFS tree, optimal clusters are identified based on the bound k and on node separator size.

6.3.7 Summary

We have presented a hybrid algorithm that uses a customizable amount of memory and guarantees optimality. The algorithm uses cycle cuts to guarantee memory-boundedness and caching between clusters to reduce the complexity. The algorithm is particularly efficient on loose problems, where most areas are explored with a linear number of messages (like in DPOP), and only small, tightly connected components are explored using the less efficient bounded inference. This means that the large overheads associated with the sequential exploration can be avoided in most parts of the problem.

Experimental results on three problem domains show that this approach gives good results for low

width, practically sized optimization problems. MB-DPOP consistently outperforms the previous state of the art in DCOP (ADOPT) with respect to 3 metrics. In our experiments, we have observed speedups of up to 5 orders of magnitude.

6.4 O-DPOP: Message size vs. Number of Messages

In this section we propose O-DPOP, a new distributed algorithm for DCOP that can also be applied to *open* constraint optimization problems (OCOP), i.e. problems that feature unbounded domains[70]. The O-DPOP algorithm explores the same search space as DPOP or ADOPT[141], but does so in an incremental, best-first fashion suitable for open problems.

As seen in Chapter 3, complete algorithms for distributed constraint optimization fall in two main categories: search (see[39, 96, 141, 198, 225]), and dynamic programming (see[107, 160]).

On one hand, search algorithms (e.g. ADOPT) require linear memory and message size, and the worst case complexity can sometimes be avoided if effective pruning is possible. However, they produce an exponential number of small messages, which typically entails large networking overheads.

On the other hand, dynamic programming algorithms (e.g. DPOP) have the important advantage that they produce fewer messages, therefore less overhead. DPOP for example requires a linear number of messages. The disadvantage is that the maximal message size and memory requirements grows exponentially in the induced width of the constraint graph. Furthermore, the worst case complexity is always incurred.

In this section we introduce O-DPOP, a hybrid which combines some advantages of both worlds: First, it uses messages whose size only grows linearly (as in search) with the treewidth of the problem. Second, by letting agents explore values in a best-first order, it avoids incurring always the worst case complexity as DPOP, and on average it saves a significant amount of computation and information exchange. This is possible because the agents in O-DPOP use a best-first order for value exploration, and an optimality criterion that allows them to prove optimality even without exploring all the values of their parents. This makes O-DPOP applicable also to open constraint optimization problems, where variables may have unbounded domains[70].

We describe next the O-DPOP algorithm (Section 6.4.1 and Section 6.4.2), show examples, and evaluate its complexity, both theoretically (Section 6.4.3) and experimentally (Section 6.4.4). Although its worst case complexity is the same as for DPOP, O-DPOP exhibits in our experiments significant savings in computation and information exchange.

O-DPOP is described in Algorithm 11. It works in 3 phases:

1. Phase 1 - a **DFS traversal**, as in DPOP (see Figure 6.5 for an example DFS).
2. Phase 2 - (**ASK/GOOD**) phase, which is a replacement of the UTIL phase from DPOP. It is an iterative, bottom-up utility propagation process, where each node repeatedly asks (via *ASK* messages) its children for valuations (*goods*) until it can compute suggested optimal values for its ancestors included in its separator. It then sends these goods to its parent. This phase finishes when the root received enough valuations to determine its optimal value.

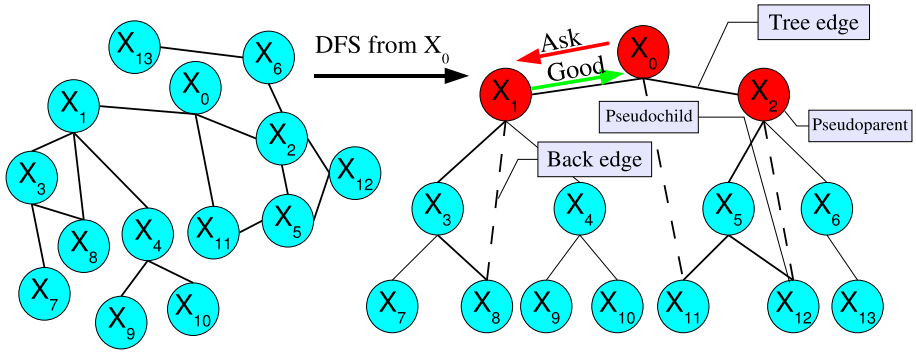


Figure 6.5: *A problem graph and a rooted DFS tree. ASK messages go top-down, and GOOD messages (valued goods) go bottom-up. All messages are of linear size.*

Algorithm 11 *O-DPOP - Open/Distributed Optimization*

O-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): each agent X_i does:

DFS arrangement: run token passing Algorithm 3

1 At completion, X_i knows $P_i, PP_i, C_i, PC_i, Sep_i$

Main process

2 $sent\_goods \leftarrow \emptyset$

3 **if** X_i *is root* **then**

ASK/GOOD until valuation sufficiency

4 **else**

5 **while** *!received VALUE message* **do**

6 Process incoming ASK and GOOD messages

Process ASK

7 **while** *!sufficiency conditional on* $sent\_goods$ **do**

8 select C_i^{ask} among C_i

9 send ASK message to all C_i^{ask}

10 wait for GOOD messages

11 find $best\_good \in Sep_i$ s.t. $best\_good \notin sent\_goods$

12 add $best\_good$ to $sent\_goods$, and send it to P_i

Process GOOD($\mathbf{gd}, \mathbf{X_k}$)

13 add gd to $goodstore(X_k)$

14 check for conditional sufficiency

---

3. Phase 3 - **VALUE propagation** as in DPOP

6.4.1 O-DPOP Phase 2: *ASK/GOOD* Phase

In backtracking algorithms, the control strategy is top-down: starting from the root, the nodes perform assignments and inform their children about these assignments. In return, the children determine their best assignments given these decisions, and inform their parents of the utility or bounds on this utility.

This top-down exploration of the search space has the disadvantage that the parents make decisions about their values blindly, and need to determine the utility for every one of their values before deciding on the optimal one. This can be a very costly process, especially when domains are large.

Additionally, if memory is bounded, many utilities have to be derived over and over again[141, 170]. This, coupled with the asynchrony of these algorithms makes for a large amount of effort to be duplicated unnecessarily[241].

6.4.1.1 Propagating GOODs

In contrast, we propose a bottom-up strategy in O-DPOP, similar to the one of DPOP. In this setting, higher nodes do not assign themselves values, but instead ask their children what values would they prefer. Children answer by proposing values for the parents' variables. Each such proposal is called a *good*, and has an associated utility that can be achieved by the subtree rooted at the child, in the context of the proposal.

Definition 22 (Good) *Given a node X_i, its parent P_i and its separator Sep_i, a* **good message** $GOOD_i^{P_i}$ *sent from X_i to P_i is a tuple $\langle assignments, utility\rangle$ as follows: $GOOD_i^{P_i} = \langle\{X_j = v_j^k | X_j \in Sep_i, v_j^k \in D_j\}, v \in \mathbb{R}\rangle$.*

In words, a good $GOOD_i^{P_i}$ sent by a node X_i to its parent P_i has exactly one assignment for each variable in Sep_i, plus the associated utility generated by this assignment for the subtree rooted at X_i. In the example of Figure 6.5, a good sent from X_5 to X_2 might have this form: $GOOD_5^2 = \langle X_2 = a, X_0 = c, 15\rangle$, which means that if $X_2 = a$ and $X_0 = c$, then the subtree rooted at X_5 gets 15 units of utility.

Definition 23 (Compatibility: $\equiv$) *Two good messages $GOOD_1$ and $GOOD_2$ are* **compatible** *(we write this $GOOD_1 \equiv GOOD_2$) if they do not differ in any assignment of the shared variables. Otherwise, $GOOD_1 \not\equiv GOOD_2$.*

Example: $\langle X_2 = a, X_0 = c, 15\rangle \equiv \langle X_2 = a, 7\rangle$, but $\langle X_2 = a, X_0 = c, 15\rangle \not\equiv \langle X_2 = b, 7\rangle$.

Definition 24 (Join: $\oplus$) *The* **join** $\oplus$ *of two* compatible *good messages $GOOD_j^i = \langle assig_j, val_j\rangle$ and $GOOD_k^i = \langle assig_k, val_k\rangle$ is a new good $GOOD_{j,k}^i = \langle assig_j \cup assig_k, val_j + val_k\rangle$*

Example in Figure 6.5: let $GOOD^5_{11} = \langle X_5 = a, X_0 = c, 15\rangle$ and $GOOD^5_{12} = \langle X_5 = a, X_2 = b, 7\rangle$. Then $GOOD^5_{11} \oplus GOOD^5_{12} = \langle X_2 = b, X_0 = c, X_5 = a, 22\rangle$.

6.4.1.2 Value ordering and bound computation

Any child X_j of a node X_i delivers to its parent X_i a sequence of $GOOD^i_j$ messages that explore different combinations of values for the variables in Sep_j, together with the corresponding utilities. We introduce the following important assumption:

> **Best-first Assumption**: leaf nodes (without children) report their $GOOD$s in order of non-increasing utility.

This assumption is easy to satisfy in most problems: it corresponds to ordering entries in a relation according to their utilities. Similarly, agents usually find it easy to report what their most preferred outcomes are.

We now show a method for propagating $GOOD$s so that all nodes always report $GOOD$s in order of non-increasing utility provided that their children follow this order. Together with the assumption above, this will give an algorithm where the first $GOOD$ generated at the root node is the optimal solution. Furthermore, the algorithm will be able to generate this solution without having to consider all value combinations.

Consider thus a node X_i that receives from each of its children X_j a stream of $GOOD$s in an asynchronous fashion, but in non-increasing order of utility.

Notation: let $LAST^i_j$ be the last good sent by X_j to X_i. Let $\langle Sep_i\rangle$ be the set of all possible instantiations of variables in Sep_i. A tuple $s \in \langle Sep_i\rangle$ is such an instantiation. Let $GOOD^i_j(t)$ be a good sent by X_j to X_i that is compatible with the assignments in the tuple t.

Based on the goods that X_j has already sent to X_i, one can define lower (LB) and upper (UB) bounds for each instantiation $s \in \langle Sep_i\rangle$:

$$LB^i_j(s) = \begin{cases} val(GOOD^i_j(t)) & \text{if } X_j \text{ sent } GOOD^i_j(t) \text{ s.t. } t \equiv s \\ -\infty & \text{otherwise} \end{cases}$$

$$UB^i_j(s) = \begin{cases} val(GOOD^i_j(t)) & \text{if } X_j \text{ sent } GOOD^i_j(t) \text{ s.t. } t \equiv s \\ val(LAST^i_j) & \text{if } X_j \text{ has sent any } GOOD^i_j \\ +\infty & \text{if } X_j \text{ has not sent any } GOOD^i_j \end{cases}$$

The influence of all children of X_i is combined in upper and lower bounds for each $s \in \langle Sep_i \rangle$ as follows:

- $UB^i(s) = \sum_{X_j \in C_i} UB^i_j(s)$; if any of $X_j \in C_i$ has not yet sent any good, then $UB^i_j(s) = +\infty$, and $UB^i(s) = +\infty$. $UB^i(s)$ is the maximal utility that the instantiation s could possibly have for the subproblem rooted at X_i, no matter what other goods will be subsequently received by X_i. Note that it is possible to infer an upper bound on the utility of any instantiation $s \in \langle Sep_i \rangle$ as soon as even a single $GOOD$ message has been received from each child. This is the result of the assumption that $GOOD$s are reported in order of non-increasing utility.
- $LB^i(s) = \sum_{X_j \in C_i} LB^i_j(s)$; if any of $X_j \in C_i$ has not yet sent any good compatible with s, then $LB^i_j(s) = -\infty$, and $LB^i(s) = -\infty$. $LB^i(s)$ is the minimal utility that the tuple $s \in \langle Sep_i \rangle$ could possibly have for the subproblem rooted at X_i, no matter what other goods will be subsequently received by X_i.

Examples based on Table 6.2:

- $GOOD^4_{10}(X_4 = c) = \langle [X_4 = c], 4 \rangle$.
- $LAST^4_{10} = \langle [X_4 = a], 3 \rangle$.
- $LB^4_{10}(X_4 = c) = 4$ and $LB^4_9(X_4 = c) = -\infty$, because X_4 has received a $GOOD^4_{10}(X_4 = c)$ from X_{10}, but not a $GOOD^4_9(X_4 = c)$ from X_9.
- Similarly, $UB^4_{10}(X_4 = c) = 4$ and $UB^4_9(X_4 = c) = val(LAST^4_9) = val(GOOD^4_9(X_4 = f)) = 1$, because X_4 has received a $GOOD(X_4 = c)$ from X_{10}, but not from X_9, so the latter is replaced by the latest received good.

6.4.1.3 Valuation-Sufficiency

In DPOP, agents receive all $GOOD$s grouped in single messages. In O-DPOP, $GOOD$s can be sent individually and asynchronously as long as the order assumption is satisfied. Therefore, X_i can determine when it has received **enough** goods from its children in order to be able to determine the next best combination of values of variables in Sep_i[70]. In other words, X_i can determine when any additional goods received from its children X_j will not matter w.r.t. the choice of optimal tuple for Sep_i. X_i can then send its parent P_i a valued good $t^* \in Sep_i$ suggesting this next best value combination.

Definition 25 *Given a subset S of tuples from $\langle Sep_i \rangle$, a tuple $t^* \in \{\langle Sep_i \rangle \backslash S\}$ is* dominant conditional on the subset S, *when* $\forall t \in \{\langle Sep_i \rangle \setminus S | t \neq t^*\}, LB(t^*) > UB(t)$.

In words, t^* is the next best choice for Sep_i, after the tuples in S. This can be determined once there have been received enough goods from children to allow the finding that one tuple's lower bound is greater than all other's upper bound. Then the respective tuple is conditional-dominant.

Definition 26 *A variable is* valuation-sufficient conditional on a subset $S \subset \langle Sep_i \rangle$ of instantiations of the separator *when it has a tuple t^* which is dominant conditional on S.*

6.4.1.4 Properties of the Algorithm

The algorithm used for propagating $GOOD$s in O-DPOP is given by process ASK in Algorithm 11. Whenever a new $GOOD$ is asked by the parent, X_i repeatedly asks its children for $GOOD$s. In response, it receives $GOOD$ messages that are used to update the bounds. These bounds are initially set to $LB^i(\forall t) = -\infty$ and $UB^i(\forall t) = +\infty$. As soon as at least one message has been received from all children for a tuple t, its upper bound is updated with the sum of the utilities received. As more and more messages are received, the bounds become tighter and tighter, until the lower bound of a tuple t^* becomes higher than the upper bound of any other tuple.

At that point, we call t^* *dominant*. X_i assembles a good message $GOOD_i^{P_i} = \langle t^*, val = LB^i(t^*) = UB^i(t^*) \rangle$, and sends it to its parent P_i. The tuple t^* is added to the $sent\_goods$ list.

Subsequent ASK messages from P_i will be answered using the same principle: gather goods, recompute upper/lower bounds, and determine when another tuple is dominant. However, the dominance decision is made while ignoring the tuples from $sent\_goods$, so the "next-best" tuple will be chosen. This is how it is ensured that each node in the problem will receive utilities for tuples *in decreasing order of utility* i.e. in a best-first order, and thus we have the following Theorem:

Proposition 10 (Best-first order) *Provided that the leaf nodes order their relations in non-increasing order of utility, each node in the problem sends $GOOD$s in the non-increasing order of utility i.e. in a best-first order.*

PROOF. By assumption, the leaf nodes send $GOOD$s in best-first order. Assume that all children of X_i satisfy the Theorem. Then the algorithm correctly infers the upper bounds on the various tuples, and correctly decides conditional valuation-sufficiency. If it sends a $GOOD$, it is conditionally dominant given all $GOOD$s that were sent earlier, and so it cannot have a lower utility than any $GOOD$ that might be sent later. □

Example 15 (Conditional valuation-sufficiency: an example) *Let us consider a possible execution of O-DPOP on the example problem from Figure 6.5. Let us consider the node X_4, and let the relation r_4^1 be as described in Table 6.1.*

$X_1/X_4 =$	a	b	c	d	e	f
$X_1 = a$	1	2	6	2	1	2
$X_1 = b$	5	1	2	1	2	1
$X_1 = c$	2	1	1	1	2	1

Table 6.1: Relation $R(X_4, X_1)$.

X_9	X_{10}	X_1
$\langle \mathbf{X_4 = a, 6} \rangle$	$\langle X_4 = b, 5 \rangle$	$\langle X_4 = c, X_1 = a, 6 \rangle$
$\langle X_4 = d, 5 \rangle$	$\langle X_4 = c, 4 \rangle$	$\langle \mathbf{X_4 = a, X_1 = b, 5} \rangle$
$\langle X_4 = f, 1 \rangle$	$\langle \mathbf{X_4 = a, 3} \rangle$	$\langle X_4 = b, X_1 = a, 2 \rangle$
$\vdots$	$\vdots$	$\vdots$

Table 6.2: Goods received by X_4. The relation r_4^1 is present in the last column, sorted best-first.

As a result to its parent X_1 asking X_4 for goods, let us assume that X_4 has repeatedly requested goods from its children X_9 and X_{10}. X_9 and X_{10} have replied each with goods; the current status is as described in Table 6.2.

In addition to the goods obtained from its children, X_4 has access to the relation r_4^1 with its parent, X_1. This relation will also be explored in a best-first fashion, exactly as the tuples received from X_4's children (see Table 6.2, last column).

Let us assume that this is the first time X_1 has asked X_4 for goods, so the $sent\_goods$ list is empty. We compute the lower and upper bounds as described in the previous section. We obtain that $LB^i(\langle X_4 = a, X_1 = b \rangle) = 14$. We also obtain that $\forall t \neq \langle X_4 = a, X_1 = b \rangle$, $UB^i(t) < LB^i(\langle X_4 = a, X_1 = b \rangle) = 14$. Therefore, $\langle X_4 = a, X_1 = b \rangle$ satisfies the condition from Definition 26 and is thus dominant conditional on the current $sent\_goods$ set (which is empty). Thus, X_4 records $\langle X_4 = a, X_1 = b, 14 \rangle$ in $sent\_goods$ and sends $GOOD(X_1 = b, 14)$ to X_1.

Should X_1 subsequently ask for another good, X_4 would repeat the process, this time ignoring the previously sent tuple $GOOD(X_1 = b, 14)$.

6.4.1.5 Comparison with the *UTIL* phase of *DPOP*

In DPOP, the separator Sep_i of a node X_i gives the set of dimensions of the *UTIL* message from X_i to its parent: $Sep_i = dims(UTIL_i^{P_i})$ Therefore, the size of a *UTIL* message in DPOP is $d^{|Sep_i|}$, where d

is the domain size. This results in memory problems in case the induced width of the constraint graph is high.

In O-DPOP, the *ASK/GOOD* phase is the analogue of the *UTIL* phase from *DPOP*. A $GOOD_i^{P_i}$ message corresponds exactly to a single utility from a $UTIL_i^{P_i}$ message from DPOP, and has the same semantics: it informs P_i how much utility the whole subtree rooted at X_i obtains when the variables from Sep_i take that particular assignment.

The difference is that the utilities are sent on demand, in an incremental fashion. A parent P_i of a node X_i sends to X_i an *ASK* message that instructs X_i to find the next best combination of values for the variables in Sep_i, and compute its associated utility. X_i then performs a series of the same kind of queries to its children, until it gathers enough goods to be able to determine this next best combination $t^* \in \langle Sep_i \rangle$ to send to P_i. At this point, X_i assembles a message $GOOD_i^{P_i}(t^*, val)$ and sends it to P_i.

6.4.2 O-DPOP Phase 3: top-down VALUE assignment phase

The *VALUE* phase is similar to the one from *DPOP*. Eventually, the root of the DFS tree becomes *valuation-sufficient*, and can therefore determine its optimal value. It initiates the top-down *VALUE* propagation phase by sending a *VALUE* message to its children, informing them about its chosen value. Subsequently, each node X_i receives the $VALUE_{P_i}^i$ message from its parent, and determines its optimal value as follows:

1. X_i searches through its $sent\_list$ for the first good $GOOD^{i*}$ (highest utility) compatible with the assignments received in the *VALUE* message.
2. X_i assigns itself its value from $GOOD^{i*}$: $X_i \leftarrow v_i^*$
3. $\forall X_j \in C_i$, X_i builds and sends a *VALUE* message that contains $X_i = v_i^*$ and the assignments shared between $VALUE_{P_i}^i$ and Sep_j. Thus, X_j can in turn choose its own optimal value, and so on recursively to the leaves.

6.4.3 O-DPOP: soundness, termination, complexity

Theorem 3 (Soundness) *O-DPOP is sound.*

PROOF. O-DPOP combines goods coming from independent parts of the problem (subtrees in DFS are independent). Theorem 10 shows that the goods arrive in the best-first order, so when we have valuation-sufficiency, we are certain to choose the optimal tuple, provided the tuple from Sep_i is optimal.

The top-down *VALUE* propagation ensures (through induction) that the tuples selected to be parts of the overall optimal assignment, are indeed optimal, thus making also all assignments for all Sep_i optimal. □

Theorem 4 (Termination) *O-DPOP terminates in at most $(h-1) \times d^w$ synchronous ASK/GOOD steps, where h is the depth of the DFS tree, d bounds the domain size, and w is the width of the chosen DFS. Synchronous here means that all siblings send their messages at the same time.*

PROOF. The longest branch in the DFS tree is of length $h-1$ (and h is at most n, when the DFS is a chain). Along a branch, there are at most d^{Sep_i} *ASK/GOOD* message pairs exchanged between any node X_i and its parent. Since $Sep_i \leq w$, it follows that at most $(h-1) \times d^w$ synchronous *ASK/GOOD* message pairs will be exchanged. □

Theorem 5 (Complexity) *The number of messages and memory required by O-DPOP is $O(d^w)$.*

PROOF. By construction, all messages in O-DPOP are linear in size. Regarding the number of messages:

1. the DFS construction phase produces a linear number of messages: $2 \times m$ messages (m is the number of edges);
2. the *ASK/GOOD* phase is the analogue of the *UTIL* phase in DPOP. The worst case behavior of O-DPOP is to send sequentially the contents of the *UTIL* messages from DPOP, thus generating at most d^w *ASK/GOOD* message pairs between any parent/child node (d is the maximal domain size, and w is the induced width of the problem graph). Overall, the number of messages is $O((n-1) \times d^w)$. Since all these messages have to be stored by their recipients, the memory consumption is also at most d^w.
3. the *VALUE* phase generates $n-1$ messages, (n is the number of nodes) - one through each tree-edge.

□

Notice that the d^w complexity is incurred only in the worst case. Consider an example: a node X_i receives first from all its children the same tuple as their most preferred one. Then this is simply chosen as the best and sent forward, and X_i needs only linear memory and computation!

6.4.4 Experimental Evaluation

We experimented with distributed meeting scheduling in an organization with a hierarchical structure (a tree with departments as nodes, and a set of agents working in each department). The CSP model is

Agents	10	20	30	50	100
Meetings	3	9	11	19	39
Variables	10	31	38	66	136
Constraints	10	38	40	76	161
# of messages	35 / 9	778 / 30	448 / 37	3390 / 65	9886 / 135
Max message size	1 / 100	1 / 1000	1 / 100	1 / 1000	1 / 1000
Total Goods	35 / 360	778 / 2550	448/1360	3390 / 10100	9886 / 16920

Table 6.3: O-DPOP vs DPOP tests on meeting scheduling (values stated as O-DPOP / DPOP)

the PEAV model from[127]. Each agent has multiple variables: one for the start time of each meeting it participates in, with 10 timeslots as values. Mutual exclusion constraints are imposed on the variables of an agent, and equality constraints are imposed on the corresponding variables of all agents involved in the same meeting. Private, unary constraints placed by an agent on its own variables show how much it values each meeting/start time. Random meetings are generated, each with a certain utility for each agent. The objective is to find the schedule that maximizes the overall utility.

Table 6.3 shows how our algorithm scales up with the size of the problems. All experiments are run on the FRODO multiagent simulation platform[154]. The values are depicted as O-DPOP / DPOP, and do not include the DFS and VALUE messages (identical). The number of messages refers to *ASK/GOOD* message pairs in $O-DPOP$ and *UTIL* messages in $DPOP$. The maximal message size shows how many utilities are sent in the largest message in *DPOP*, and is always 1 in O-DPOP (a single good sent at a time). The last row of the table shows significant savings in the number of utilities sent by O-DPOP (*GOOD* messages) as compared to DPOP (total size of the *UTIL* messages).

6.4.5 Comparison with search algorithms

In backtrack search algorithms, the control strategy is top-down: starting from the root, the agents perform assignments and inform their children about these assignments. In return, the children determine their best assignments given these decisions, and inform their parents of the utility or bounds on this utility. This top-down exploration of the search space has the disadvantage that the parents make decisions about their values blindly, and need to determine the utility for every one of their values before deciding on the optimal one. This can be a very costly process, especially when domains are large. Additionally, if memory is bounded, many utilities have to be derived over and over again[141, 170]. This, coupled with the asynchrony of these algorithms makes for a large amount of effort to be duplicated unnecessarily[241].

In contrast, O-DPOP uses a bottom-up strategy, similar to the one of DPOP. In this setting, higher

agents do not assign themselves values, but instead ask their children what values would they prefer. Children answer by proposing values for the parents' variables. These proposals are similar to the COST messages in search algorithms, the difference being that they are sent proactively, and in the context chosen by the lower agents, as opposed to search, where the proposals are chosen by the higher agents. By using the idea of *valuation sufficiency*, O-DPOP can possibly find the optimal solution without exploring all values of some of the variables, which is in contrast with search algorithms. This also enables O-DPOP to be able to deal with *open* problems, i.e. problems with unbounded domains.

6.4.6 Summary

O-DPOP uses linear size messages by sending the utility of each tuple separately. Based on the best-first assumption, we use the principle of open optimization[70] to incrementally propagate these messages even before the utilities of all input tuples have been received. This can be exploited to significantly reduce the amount of information that must be propagated. In fact, the optimal solution may be found without even examining all values of the variables, thus being possible to deal with unbounded domains.

Preliminary experiments on distributed meeting scheduling problems show that O-DPOP gives good results when the problems have low induced width.

As the new algorithm is a variation of DPOP, we can apply to it the techniques for self-stabilization[165], approximations and anytime solutions[158], distributed implementation and incentive-compatibility[171] that have been proposed for DPOP.

Chapter 7

Tradeoffs between Memory/Message Size and Solution Quality

In this chapter we discuss possible tradeoffs between solution quality on one hand, and computation/memory/communication requirements on the other hand. We introduce two algorithms that offer configurable tradeoffs quality/effort.

In Section 7.1, we introduce LS-DPOP(k), a hybrid algorithm which is a mix between classical local search methods in which nodes take decisions based only on local information, and full inference methods that guarantee completeness. LS-DPOP operates in the framework from Section 6.2 for detecting difficult subproblems, where normal DPOP cannot be applied. In such subproblems, LS-DPOP executes a local search procedure guided by as much inference as allowed by k. LS-DPOP(k) can be seen as a large neighborhood search, where exponential neighborhoods are rigorously determined according to problem structure, and polynomial efforts are spent for their complete exploration at each local search step.

The second contribution of this chapter is A-DPOP (Section 7.2), a parameterized approximation scheme based on DPOP, which allows the desired tradeoff between solution quality and computational complexity. A-DPOP allows to adapt the size of the largest message to the desired approximation ratio. Clusters of high width are detected as in Section 6.2 and explored with approximate propagations using the idea of minibuckets[49, 51].

7.1 LS-DPOP: a local search - dynamic programming hybrid

We present a new hybrid algorithm for local search in distributed combinatorial optimization. This method is a mix between classical local search methods in which nodes take decisions based only on local information, and full inference methods that guarantee completeness.

We propose LS-DPOP(k), a hybrid method that combines the advantages of both these approaches.

LS-DPOP(k) is a utility propagation algorithm controlled by a parameter k which specifies the maximal allowable amount of inference. The maximal space requirements are exponential in this parameter. In the dense parts of the problem, where the required amount of inference exceeds this limit, the algorithm executes a local search procedure guided by as much inference as allowed by k. LS-DPOP(k) can be seen as a large neighborhood search, where exponential neighborhoods are rigorously determined according to problem structure, and polynomial efforts are spent for their complete exploration at each local search step.

For difficult optimization problems, local search methods have been developed. These methods start with a random assignment, and then gradually improve it by applying incremental changes. Their advantage is that they require linear memory, and in many cases provide good solutions with a small amount of effort. However, the decisions taken are often myopic in the sense that they take into account only local information, thus getting stuck into local optima rather easily. Large neighborhood search[3] tries to overcome this problem by exploring a much larger set of neighboring states before moving to the next one. Dynamic programming has already been recognized as an efficient way to explore exponential size neighborhoods with a polynomial effort[67]. Another example of such a hybrid technique is the work of Kask and Dechter from[105] (see Section 7.1.5).

For distributed environments, there are distributed local search methods like DSA ([109]) / DBA([237]) for optimization, and DBA for satisfaction ([226]). To our knowledge, the concept of large neighborhoods has not been exploited in distributed environments.

We propose a distributed algorithm that combines the advantages of both these approaches. This method is a utility propagation algorithm controlled by a parameter k which specifies the maximal allowable amount of inference. The maximal space requirements are exponential in this parameter. In the dense parts of the problem, where the required amount of inference exceeds this limit, the algorithm executes a local search procedure guided by as much inference as allowed by k. If this parameter is equal to the induced width of the graph or larger, then the algorithm is full inference, therefore complete. Larger values of k are conjectured to produce better results.

We show the efficiency of this approach with experimental results from the distributed meeting scheduling domain.

The rest of this chapter is structured as follows: Section 7.1.1 presents the hybrid optimization algorithm. Section 7.1.4 presents an experimental evaluation. Section 7.1.5 presents the relationship between this approach and existing work. Section 7.1.6 concludes.

7.1.1 *LS-DPOP* - local search/inference hybrid

We keep the basic utility propagation mechanism from DPOP, but we introduce a control parameter k which specifies the maximal amount of inference (maximal message dimensionality). In the dense

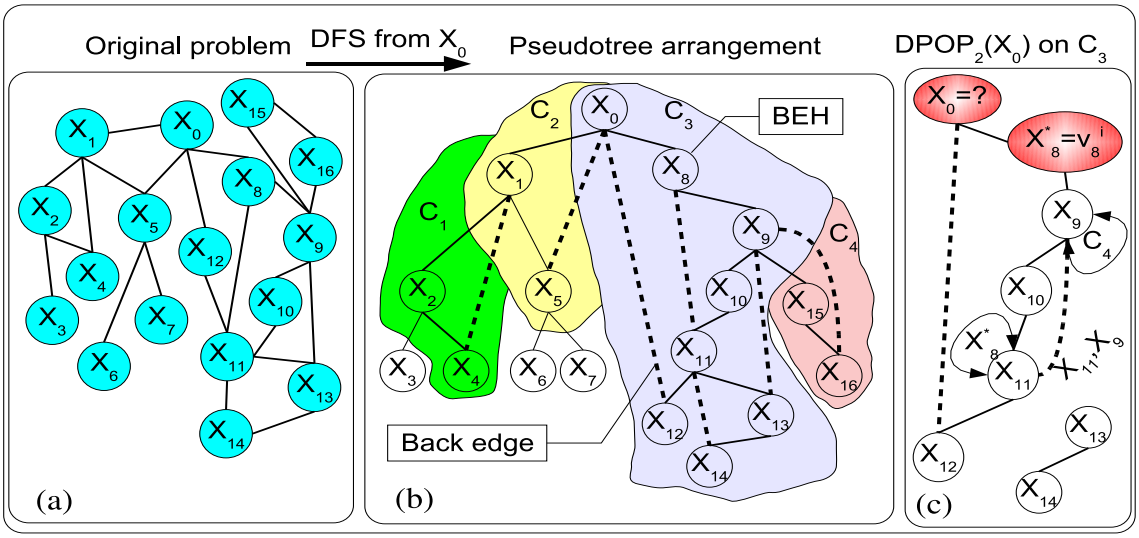


Figure 7.1: *A problem graph, one possible rooted DFS tree, and an execution detail of DPOP in C_3.*

parts of the problem, the exact propagation produces messages with more dimensions than this limit. In such cases, the algorithm executes a local search procedure guided by as much inference as allowed by k. The nodes whose processing by inference would exceed the k limit are the ones who execute the local search procedure. All other nodes execute the normal utility propagation protocol.

7.1.1.1 Detecting areas where local search is required

During the utility propagation procedure from *DPOP*, each node computes the *UTIL* message for its parent. In high width areas, some nodes have to send messages whose dimensionality exceeds k. In such cases, those nodes choose $dims - k$ dimensions of the message, mark them as *local search* dimensions, project them out of the outgoing message, and add these dimensions to the *context* of the message. Thus, the final dimensionality of the message is k (size limit observed). The dimensions to be marked as LS are chosen according to their level in the pseudotree. This is easy to determine for each node just by finding their position in the node's root path.

Example 16 *For example, consider C_3 in Figure 7.1(b). If we run LS-DPOP with $k = 2$, then the messages $UTIL_{12}^{11}$ and $UTIL_{13}^{11}$ proceed normally as in DPOP, with $dims(UTIL_{12}^{11}) = \{11, 0\}$ and $dims(UTIL_{13}^{11}) = \{11, 9\}$. However, $dims(UTIL_{11}^{10}) = \{10, 0, 8, 9\}$, thus it exceeds $k = 2$. Therefore, X_{11} marks X_0 and X_8 (the 2 highest nodes in $dims(UTIL_{11}^{10})$) as LS nodes, projects them out of $UTIL_{11}^{10}$, and adds them to the context of $UTIL_{11}^{10}$. Thus, $dims(UTIL_{11}^{10}) = \{10, 9\}$ and $context(UTIL_{11}^{10}) = \{0^*, 8^*\}$.*

The propagation continues, and when the respective messages arrive at X_8 and X_0, they know that

they must revert to local search. Note that in this example, X_0 is labeled as LS only in C_3, and not in C_2 (k not exceeded in C_2), so it will receive an exact message from C_2, and it will perform local search in C_3, together with X_8.

7.1.1.2 Local search in independent clusters

In the example of Figure 7.1, we notice that there are 4 independent parts which do not communicate between themselves except for some "frontier" nodes. These 4 cyclic subgraphs ($C_1 - C_4$), separated by the nodes X_0, X_1, X_9 can be explored separately for optimal solutions, and then the results assembled through the same *UTIL/VALUE* propagations. The advantage of this separation becomes apparent if we consider that many such separate problem components could be too complex to apply the exact *DPOP* propagation, and it may be needed to apply the local search mechanism. Then, it is obvious that by applying local search on each independent component C_t separately, we restrict the search space that needs to be explored from $d^{|LS|}$ to $d^{|LS(C_t)|}$, where $|LS|$ is the total number of *LS* nodes in the whole problem, and $|LS(C_t)|$ is the number of *LS* nodes in the component C_t. This, together with optimal combination of these local optima through *UTIL/VALUE* propagations, gives us a much better chance of finding a better overall local optima.

Identifying these frontier nodes is easy using the following definition:

Definition 27 (Width of a tree edge) *We define the **width** of an edge as follows: 0 if the edge is a back edge; if the edge is a tree edge, its width is the number of back edges with distinct handlers that include this edge in their associated tree paths.*

Please note that this definition coincides with the dimensionality of the *UTIL* message that travels through this edge in DPOP. A node is a frontier node for a subgraph if the message it receives from its child contains only itself as dimension/context. For example, X_9 is a frontier node for C_4 because $UTIL^9_{15}$ contains only itself as dimension ($X_9 - X_{15}$ has width 1). X_9 is not a frontier node for the subgraph rooted at X_{10} because $UTIL^9_{10}$ has X_9, X_8, X_0 as dimensions/context ($X_9 - X_{10}$ has width 3). This classification is determined at run time based on the *UTIL* messages received from children.

If a frontier node is also designated a LS node in one of its subtrees, then that node will send its *UTIL* message to its parent only after having explored through local search the respective subtree. For example, assume C_4 hanging out from X_9 would be so complex as to require local search. Then X_9 would be marked as *LS*, and it would first participate in the local search in C_4, and only after a local optimum is reached there, would it start its propagation(s) in C_3. The utilities computed as the local optima for each of its values in C_4 are then added to the messages going through C_3. The process is logically equivalent to replacing C_4 with a unary constraint on X_9.

7.1.1.3 One local search step

In the subgraphs where local search is required, the *LS* nodes start by assigning themselves values. Then, we can run a DPOP-like propagation on the cyclic subgraph for each *LS* node X_n. For each propagation, we consider all LS nodes assigned with their current values, except for X_n. Such a propagation is just a simple variation of the $DPOP$ one, where instead of applying projections for all nodes, we execute slices for the nodes in the LS except X_n. Thus, X_n can determine how much utility each one of its values gives for the whole cyclic subgraph in which it is involved, *provided the other LS nodes maintain their current values*. It does so by joining all incoming $UTIL$ messages, and projecting out any other dimensions than itself. The result is a vector (one dimension) with the desired valuations. The value giving the maximal valuation can be proposed as the next value (in case it is different than the current value).

Figure 7.1.(c) shows an example execution of a local search step for X_0. All *LS* nodes send to their pseudochildren value messages, announcing their current values. The propagation starts normally from the leaves (X_{12} sends X_{11} a message with X_{11} and X_0 as dimensions). X_{11} performs normally the join between the messages it received from its children. Note that the message it received initially from X_{13} can be reused, since there is no link in that subtree with any LS node. Additionally, since X_8 is considered fixed at its present value, the relation $X_8 - X_{11}$ is logically replaced by a corresponding unary constraint on X_{11} (this is the slice of R^8_{11} along the current value of X_8, computed by X_{11}). The join is performed also with this induced unary constraint, and the relation R^{10}_{11}. $X11$ projects itself out of the join, and sends the message to X_{10}. The propagation continues until X_8, which performs the join $UTIL^8_9 \oplus R^0_8$. Instead of projecting itself out of the join to compute $UTIL^0_8$, X_8 performs a slice of this join along its current value (the one previously announced to X_{11}). It then sends $UTIL^0_8$ to X_0, who receives complete information about how much each of its values is worth for the whole C_3, provided X_8 keeps its current value.

X_0 can now compute $\Delta X_0 = UTIL^0_8 \perp_{X_0} -UTIL^0_8[X_0 = v_0]$, which is the maximal improvement that the whole C_3 can achieve if X_0 changes from its current value to the new optimal one, X_8 keeps its present value, and all the other nodes in C_3 change to their new optimal values.

X_0 also initiates a top-down propagation with itself as a *LS* node. It sends X_8 $UTIL^8_0$, with $dims(UTIL^8_0) = \{X_0, X_8\}$ (actually, this message is exactly R^8_0, since X_0 does not have anything else to join for sending to X_8. R^{12}_0 is taken into account by X_{12}, when sending out $UTIL^{11}_{12}$).

X_8 joins this message with $UTIL^8_9$, and performs a slice of this join, along its current value. The result is exactly the same vector as X_0 receives from X_8 as $UTIL^0_8$. What we achieved with the uniform propagation is thus the ability of X_8 to have the same information as X_0 about the possible improvements X_0 can make if X_8 keeps the current value.

After having run all propagations (with one of the *LS* nodes being allowed to change at the time), each *LS* node X_i can thus compute ΔX_j for each other *LS* node X_j in the same cyclic subgraph. In

other words, each *LS* node X_i can thus compute the maximal improvements that each other *LS* node X_j can make, provided only X_j is allowed to change.

For the change itself, one can apply any policy known in current local search methods, and guide this policy by the Δs computed like this. The termination policy can be either a maximal number of cycles, or detection of local/global minima by detecting that all *LS* nodes have $\Delta = 0$.

Correctness In the current formulation, only the node with the highest improvement changes its value. Thus, the algorithm executes a hill climbing procedure for the nodes designated as LS, and exact inference for the rest, therefore it will reach a local maximum given by local maxima in each individual cyclic subgraph.

7.1.2 Large neighborhood exploration - analysis and complexity

Let us assume that in a cyclic subgraph C_t there are cc_t nodes designated as LS nodes, n_t total nodes, and m_t edges. The size of the neighborhood completely explored at each local search step is $cc_t \times d \times d^{n_t - cc_t}$ (for all values of each *LS* node, complete exploration of the *non-LS* nodes). The effort for each step consists of $2 \times (n_t - 1)$ *UTIL* messages sent for exploring C_t. The largest message is of size d^{k+1}. Thus, each step explores an exponential size neighborhood with a polynomial amount of effort.

Assume the termination policy for the local search process involves at most k local search steps. The whole process is then equivalent to exploring $k \times cc_t \times d \times d^{n_t - cc_t}$ neighboring states. An exhaustive search method would require at least as many messages (big communication overhead), while classical local search would not be guaranteed to completely explore this part of the search space.

7.1.3 Iterative LS-DPOP for anytime

A straightforward adaptation of LS-DPOP can be used for online solving by executing LS-DPOP iteratively with increasing bounds k, as described in Algorithm 13. Iterative LS-DPOP starts with low values for k, which means that the UTIL messages, and can be quickly computed and sent over the network. This means that a (relatively) good solution can be obtained very fast. As time goes by, executions of LS-DPOP(k) proceed, with increasing values of k, which means that the clusters of width higher than k where local search must be applied get smaller and smaller. Thus, more and more areas of the problem are explored by exact inference, and not by local search, which is expected to lead to better and better global solutions. Like this we simulate an anytime behaviour with LS-DPOP.

Remark 8 (Iterative LS-DPOP can reuse computation between iterations.) *Notice that once the threshold k exceeds the size of a node X_i's separator, and of all descendants of X_i, the UTIL message computed and sent by X_i is* exact *(i.e. it is the result of only exact inference, without any local search).*

Algorithm 12 *LS-DPOP - local search/inference hybrid.*

LS-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$, k): each agent X_i does:

***UTIL* propagation protocol**

wait for *UTIL* messages (X_k, $UTIL_k^i$) from all children $X_k \in C(i)$
if *any of* $UTIL_k^i$ *contains myself as LS node* **then** execute LS procedure
else

$JOIN_i^{P(i)} = \left(\left(\bigoplus_{c \in C(i)} UTIL_c^i\right) \oplus \left(\bigoplus_{c \in \{P(i) \cup PP(i)\}} R_i^c\right)\right)$
if X_i *is root* **then** start *VALUE* propagation
else

if $|dims(JOIN_i^{P(i)})| > k$ **then**

sort $dims(JOIN_i^{P(i)})$ by root path ($P(i)$ is always last)
mark the first $|dims(JOIN_i^{P(i)})| - k$ non-LS dimensions from the JOIN as LS, project them out and add them to the context of $JOIN_{X_i}^{P(i)}$. $P(i)$ is always kept in.

compute $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)} \perp_{X_i}$ and send it to P(i)

Local search procedure

assign a value according to heuristic (can be random)
while *termination criteria for local search not met* **do**

send *VALUE*$(X_i \leftarrow current\_value)$ messages to all $PC(i)$
wait for all corresponding *UTIL* messages to arrive
join them, and slice through $X_i \leftarrow current\_value)$; store

get and store in $agent\_view$ all *VALUE* messages ($X_k \leftarrow v_k^*$)
$v_i^* \leftarrow argmax_{X_i}\left(JOIN_{X_i}^{P(i)}[v(P(i)), v(PP(i))]\right)$
Send *VALUE*$(X_i \leftarrow v_i^*)$ to all $C(i)$ and $PC(i)$

***VALUE* propagation**$(X_k \leftarrow v_k)$
if *sending node* X_k *is pseudoparent* **then**

perform slice $R_i^k[X_k = v_k]$ and join it with *UTIL* messages from children
project self out of this join, add $X_k \leftarrow v_k$ to the context of the message and send it to parent

get and store in $agent\_view$ all *VALUE* messages ($X_k \leftarrow v_k^*$)
$v_i^* \leftarrow argmax_{X_i}\left(JOIN_{X_i}^{P(i)}[v(P(i)), v(PP(i))]\right)$
Send *VALUE*$(X_i \leftarrow v_i^*)$ to all $C(i)$ and $PC(i)$

Afterwards, for subsequent executions of LS-DPOP with larger values for k, X_i's parent P_i can simply reuse the $UTIL_i^{P_i}$ message it has previously received from X_i. Like this, X_i and its whole subtree have no more computation or message passing to do until the end of the algorithm. This effectively means that Iterative LS-DPOP explores easy (low width) parts of the problem very fast in the beginning, and then most of the work is concentrated in the difficult parts of the problem.

Algorithm 13 *Iterative LS-DPOP: Anytime based on iterative LS-DPOP*

Iterative LS-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$):

Construct DFS tree using Algorithm 3
each $X_i \in \mathcal{X}$ knows Sep_i
$w = argmax_{X_i}|Sep_i|$ (the induced width)
for $k = 1 \dots w$ **do**
    run Algorithm 9 to discover clusters of width higher than k
    run LS-DPOP(k) as follows:
    if $|Sep_i| < k$ *and* $\forall X_j$ *descendant of* X_i, $|Sep_j| < k$ **then**
        X_i *reuses* its UTIL message from LS-DPOP($k-1$) in LS-DPOP(k).
    set temporary solution according to LS-DPOP(k)

7.1.4 Experimental evaluation

Our experiments were performed on distributed meeting scheduling problems. We modeled a realistic scenario, where a set of agents working for a large organization try to jointly find the best schedule for a set of meetings. The organization itself has a hierarchical structure: a tree with departments as nodes, and a set of agents working in each department. We generate meetings with high probability within departments, and with a lower probability between agents belonging to parent-child departments.

We model this problem as a DCOP following[127]. Specifically, each agent A_i has a set of variables X_i^j, one for each meeting it is involved in. Each such variable X_i^j is controlled only by the agent A_i, and represents the time when meeting j of agent A_i will start (X_i^j has time slots t_k as values). There is an equality constraint connecting the equivalent variables of all agents involved in a particular meeting (all agents must agree on a start time for their meeting). If a meeting has k participants, it is sufficient to create $k-1$ equality constraints that connect the corresponding variables in a chain (no need to fully connect them pairwise). Since an agent cannot participate in 2 meetings at the same time, there is an all-different constraint on all variables X_i^j belonging to the same agent.

We model the utility that each agent A_i assigns to each meeting M_j at each particular time $t_k \in dom(X_i^j)$ by imposing unary constraints on the variables X_i^j; each such constraint is a vector private to A_i, and denotes how much utility A_i associates with starting meeting M_j at each time t_k. The objective is to find a schedule s.t. the overall utility is maximized.

We have run 2 series of experiments with random problems generated as specified before. In the first part, we generated "easy" problems, such that they can be solved by the complete algorithm as well, in order to see how far from the global optima the local search method is. The problems had induced width 8, and the domain size was 8, meaning the largest message in the complete algorithm has $8^8 \approx 16.5$M values. These problems are quite close to the feasibility limit for a complete algorithm.

The results of these experiments are presented in Table 7.1. Each row represents an execution with

k	LS#	%Non-LS	Cycles	Avg LS/cycle	Avg non-LS/cycle	Sol %off	Effort/step
1	68	68	11	6	13 $\rightarrow d^{13}$	10.86	640 ($O(d^2)$)
2	39	81	9	4	19 $\rightarrow d^{19}$	10.62	3072($O(d^3)$)
3	25	88	8	3	23 $\rightarrow d^{23}$	9.71	20480($O(d^4)$)
4	15	93	6	2	33 $\rightarrow d^{33}$	9.3	131072($O(d^5)$)
5	5	97	2	2	105 $\rightarrow d^{105}$	8.25	786432($O(d^6)$)
6	2	99	1	2	214 $\rightarrow d^{214}$	7.26	4194304($O(d^7)$)
∞	0	100	0	0	216 $\rightarrow d^{216}$	0.0	$O(d^8)$

Table 7.1: LS-DPOP tests: 100 agents, 59 meetings, 199 variables, 514 constraints, width 8

an increasing bound k. The columns represent (in order): the k bound, *LS#* is the total number of nodes executing the local search procedure, *%Non-LS* is the percentage of nodes executing the normal propagation, *Cycles* is the number of independent subgraphs identified, *Avg LS/Non-LS nodes per cycle* is the average number of LS/non-LS nodes in a single component, *Sol %off* is the distance from the optimal solution in percent, and *Effort/step* is an upper bound on the total amount of data transmitted within an independent component, for one local search step.

We have run the algorithm with increasing k, and noticed relatively small increases in solution quality (percent off the true optimum decreases slowly) and exponential increases of the amount of effort spent for each local search step.

We notice that small values of k are already producing good solutions, with relatively low effort. We explain this by the fact that even small values of k allow for a large percentage of nodes to execute the exact propagation, and thus at each local search step, a large exponential neighborhood is explored. For example, imposing $k = 1$ (first row in Table 7.1) still leaves on the average almost 70% of the nodes to execute the exact propagation. On the average, in a subgraph, 13 *non-LS* nodes adjust optimally to the values of the 6 *LS* nodes, which is equivalent to exploring 8^{13} neighboring states at each *LS* step.

The second sets of experiments involved much larger and more difficult instances of the same meeting scheduling problems. In this case, the problems were generated with 200 agents, 498 variables and 1405 constraints. The induced width was 20, making for a 8^{20} maximal message size, which renders DPOP completely infeasible. We ran again *LS-DPOP* with increasing k, and noticed a similar behavior: a large percentage of nodes execute exact propagation even for small k, and solution quality improves slowly with increasing k. The results are shown in Table 7.2. We conjecture that these results are close to the true optimum.

k	LS#	%Non-LS	Cycles	Avg LS/cycle	Avg non-LS/cycle	Solution	Effort/step
1	194	61	10	19	30 $\rightarrow d^{30}$	7910.0	4032($O(d^2)$)
2	131	73	10	13	36 $\rightarrow d^{36}$	7946.0	23040($O(d^3)$)
3	96	80	9	10	44 $\rightarrow d^{44}$	7964.0	139264($O(d^4)$)
4	73	85	9	8	47 $\rightarrow d^{47}$	7980.0	884736($O(d^5)$)
5	58	88	9	6	48 $\rightarrow d^{48}$	8021.0	6029312($O(d^6)$)

Table 7.2: LS-DPOP tests: 200 agents, 498 variables, 1405 constraints, width 20

7.1.5 Related Work

The nodes involved in the local search process can be thought of as *cycle cutset nodes*[51, 53]. From this perspective, there are a number of similar existing approaches.

Kask and Dechter present in[105] a method of combining a local search algorithm (GSAT) with inference. That method is formulated for constraint satisfaction problems, in a centralized setting. A subset of the problem nodes are given as cycle cutset nodes, and local search is performed on this subset. For each instantiation of the cutset nodes, a tree inference algorithm is applied to the rest of the problem. The differences between these methods are manyfold. First, our method is distributed, and is defined for optimization, not satisfaction. Second, the set of nodes that perform local search is identified at runtime (not given a priori). Third, we allow for inference with maximal width greater than 1, controlled by k. Finally, we separate the problem in distinct cyclic subgraphs which are explored separately, and the subsolutions are aggregated in a distributed fashion.

Petcu and Faltings present in[156] a distributed cycle cutset optimization method. The idea of isolating independent cyclic subgraphs appears there, too, but unfortunately there is no efficient method presented for identifying cycle cutsets nodes, nor for isolating independent cyclic subgraphs. Here, the DFS traversal of the graph is an excellent way to achieve both goals. There, exhaustive search is performed on the cycle cutset variables, as opposed to local search/propagation here. The synchronization problems between cycles from that method are solved here by simply making each node that borders 2 cyclic subgraphs wait for complete exploration of all its subtrees before sending its message to its parent.

7.1.6 Summary

We have presented the first approach to large neighborhood search in distributed optimization. Exponential neighborhoods are rigorously determined according to problem structure, and polynomial efforts are spent for their complete exploration at each local search step.

The algorithm explores independent parts of the problem simultaneously and asynchronously, and then combines the results, all in a distributed fashion. The experimental results show that this approach gives good results for low width, practically sized dynamical optimization problems. For loose problems, most of the search space is optimally explored, and only small, tightly connected components are explored by local search. This increases the chance that the algorithm avoids some of the local optima, especially for loose problems.

For future work we plan to experiment with several different value switching policies (like simultaneous switches by several variables or allowing non-improving switches) and different termination policies.

7.2 A-DPOP: approximations with minibuckets

This section introduces A-DPOP, a parameterized approximation scheme based on DPOP, which allows the desired tradeoff between solution quality and computational complexity. A-DPOP allows to adapt the size of the largest message to the desired approximation ratio. Specifically, A-DPOP can operate in two ways:

- The user can specify a parameter k, which specifies the maximal dimensionality of any UTIL message produced by the algorithm, thus effectively limiting the memory and communication requirements. In this case, A-DPOP(k) finds the best solution it can by using only $O(exp(k))$ memory.
- conversely, the user can specify a parameter δ, which specifies the maximal admitted error bound (in percent). A-DPOP(δ) then uses the least amount of computation and memory which is necessary to produce a solution which is guaranteed to be within δ % from the optimal solution.

When the optimal solution is required (i.e. $k = \infty$ or $\delta = 0$), A-DPOP reduces to DPOP, and the size of the largest message is in the worst case exponential in the width of the constraint graph. As DPOP, A-DPOP also requires only a linear number of messages in all cases.

A-DPOP(k) operates in the framework of Section 6.2 for detecting high-width clusters, where it is not possible to perform full inference as in DPOP. Clusters of high width are explored with approximate propagations using the idea of *minibuckets*[49, 51]. Specifically, every message in a high-width cluster (which would normally have more than k dimensions) is replaced with two lower dimensionality approximate messages, which contain upper-bounds and lower-bounds on utility. Therefore, in these areas of high width, A-DPOP offers a tradeoff between solution quality and required memory/communication. In areas of low width, A-DPOP uses the normal, exact DPOP propagations.

The overall behavior of A-DPOP(k) is as follows: if w is the induced width of the problem given by the chosen DFS ordering, depending on the value chosen for k, we have 3 cases:

1. If $k = 1$, only linear messages and memory are used.
2. If $k < w$, A-DPOP(k) performs exact inference in areas of width lower than k, and approximate inference in areas of width higher than k. Memory requirements are $O(exp(k))$.
3. If $k \geq w$, exact inference is done throughout the problem; A-DPOP(k) is then equivalent with DPOP (i.e. exact inference everywhere). Memory requirements are $O(exp(w))$.

A-DPOP operates in the same 3 phases as DPOP: DFS construction, UTIL propagation bottom-up (see section 7.2.1), and VALUE propagation top-down (see section 7.2.2). *A-DPOP* is formally described in Algorithm 14.

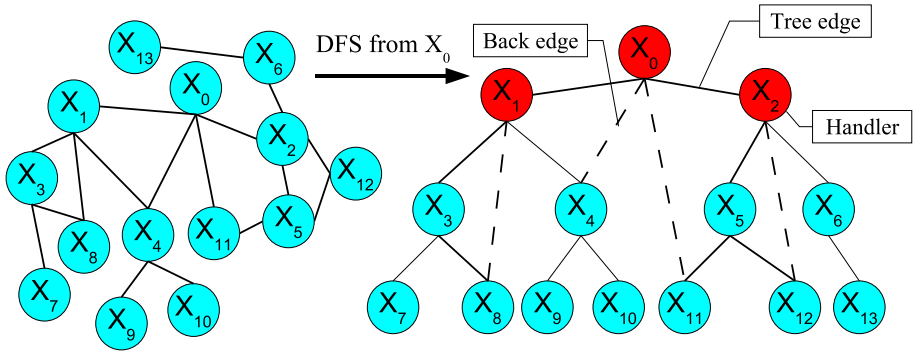


Figure 7.2: *A problem graph and a rooted DFS tree.*

Algorithm 14 *A-DPOP - Approximate Distributed Pseudotree Optimization*

A-DPOP($\mathcal{X}, \mathcal{D}, \mathcal{R}, k, \delta$): each agent X_i does:

1 **Construct DFS tree**; after completion, X_i knows P_i, PP_i, C_i, PC_i

***UTIL* propagation protocol**

2 wait for *UTIL* messages $(X_k, UTIL_k^i)$ from all children $X_k \in C_i$

3 build $JOIN_i^{P_i\pm}$ as in Equation 7.1

4 **if** X_i *is root* **then** start *VALUE* propagation

5 **else**

6 **if** $|dims(JOIN_i^{P_i\pm})| > k$ **then**

7 select $S \subset dims\left(JOIN_i^{P_i\pm}\right)$ for elimination according to section 7.2.1.1

8 compute $UTIL_i^{P_i\pm}$ as projections of $JOIN_i^{P_i\pm}$ on $S \cup X_i$, cf. equation 7.2

9 **if** $\delta(UTIL_i^{P_i\pm}) > \delta$ **then** retry with another set S; if not possible, decide for trade-off according to section 7.2.4

10 **else** $UTIL_i^{P_i\pm} = JOIN_i^{P_i\pm} \perp_{X_i}$

11 Send $UTIL_i^{P_i\pm}$ to P_i

***VALUE* propagation protocol**

12 get and store in $agent\_view$ all *VALUE* messages $(X_k \leftarrow v_k^*)$

13 compute v_i^* according to formulas 7.5 or 7.4 from section 7.2.2

14 Send *VALUE*$(X_i \leftarrow v_i^*)$ to all C_i and PC_i

7.2.1 *UTIL* propagation phase

In this section we show the modifications needed in the UTIL phase from DPOP to allow limiting the size of the UTIL messages by imposing a limit k on the maximum dimensionality. In high width areas (separator size greater than k), the algorithm drops a set $\mathcal{S}$ of dimensions to stay below the limit, and computes upper and lower bounds on utility, as detailed below.

7.2.1.1 Limiting the size of UTIL messages with approximations

In Section 4.1.2, Definition 17 we have defined the *optimal projection* operator $\perp$, which eliminates a variable from a relation by selecting the best utility for each combination of the remaining variables. This projection has the semantics of a precomputation of the optimal utility that can be achieved with the optimal values of X_k, for each instantiation of the other variables.

Definition 28 (minimal projection) *The $\perp^-$ operator (**minimal projection**): if H is a hypercube and $X_k \in dim(H)$, then $H^- = H \perp^-_{X_k}$ is a minimal projection of H along the X_k axis: for each tuple of variables in $\{dim(H) \setminus X_k\}$, all the corresponding values from H (one for each value of X_k) are tried, and the worst is chosen. The result is a hypercube with one less dimension (X_k).*

This projection has the semantics of a precomputation of lower bounds on the utility that can be achieved for each instantiation of all variables but X_k, when X_k takes its worst values. This is a guarantee that no matter what value X_k takes, the utility will not be lower than the corresponding value from H^-.

To better distinguish between the optimal projection operator $\perp$ from Section 4.1.2 and the minimal projection operator $\perp^-$ from Definition 28, we will use in the following the notation $\perp^+$ to denote the $\perp$ operator from Section 4.1.2. Notice that $\perp^-$ and $\perp^+$ are associative and commutative. Thus, a projection along a set of dimensions is identical to a sequence of projections along each dimension.

The new UTIL propagation proceeds as follows:

- as in DPOP, leaves initiate the propagation of UTIL messages, and subsequently each node computes its UTIL message and sends it to its parent.
- in areas of low width (nodes with separator sizes at most k), the nodes compute their UTIL messages normally, as in DPOP.
- in areas of high width (nodes with separator sizes at most k), every node drops from its UTIL message as many dimensions as required to observe the maximal dimensionality k, and computes approximate UTIL messages of at most k dimensions: a message with lower bounds, and a message with upper bounds (see Equations 7.1 and 7.2, and Example 17).
- upon completion, the root can determine the error bound by comparing the lower bounds with the upper bounds.

Formally, equations 7.1 and 7.2 define the approximate versions of the JOIN and UTIL hypercubes each node X_i from a high-width area would compute. The set $\mathcal{S}$ represents the set of dimensions X_i drops from its $UTIL_i^{P_i}$ message. These dimensions can be selected according to a greedy process. In[158] we have implemented this by dropping out the highest nodes in the DFS. The goal is to drop as

many dimensions as possible in order to observe the maximal dimensionality bound, without exceeding the maximal error bound. In case this is not possible, one needs to settle for a tradeoff (see section 7.2.4 for more details).

$$JOIN_i^{P_i\pm} = \left(\bigoplus_{c\in C_i} UTIL_c^{i\pm}\right) \oplus \left(\bigoplus_{p\in\{P_i\cup PP_i\}} R_i^p\right) \tag{7.1}$$

$$UTIL_i^{P_i+} = JOIN_i^{P_i+} \perp_{\mathcal{S}} \perp_{X_i}; UTIL_i^{P_i-} = JOIN_i^{P_i-} \perp_{\mathcal{S}}^{-} \perp_{X_i} \tag{7.2}$$

Example 17 *In Figure 7.2, X_4 computes $UTIL_4^1$, with $dims(UTIL_4^1) = \{X_1, X_0\}$. If $k = 1$, we have to drop $\mathcal{S} = \{X_0\}$ from $UTIL_4^1$. This is done by computing upper and lower bounds on the utility that could be achieved by X_4 and its subtree, in the best/worst case of a value of X_0. Two corresponding hypercubes, $UTIL_4^{1+} = JOIN_4^1 \perp_{X_4} \perp_{X_0}^{+}$ and $UTIL_4^{1-} = JOIN_4^1 \perp_{X_4} \perp_{X_0}^{-}$ are produced, with $dims(UTIL_4^{1+}) = dims(UTIL_4^{1-}) = \{X_1\}$. We denote by $UTIL_4^{1\pm}$ the pair ($UTIL_4^{1+}$,$UTIL_4^{1-}$).*

Let us consider a pair of 2 hypercubes H^- and H^+ with the same set of dimensions, which are lower and upper bounds on utility for each one of their tuples; to simplify notation, we denote this pair by $H^{\pm} = (H^-, H^+)$. For each tuple $\mathcal{T}$ of variables $X_j \in dims(H^{\pm})$, $H^-[\mathcal{T}]$ has the semantics of a lower bound on utility that can be achieved provided the variables in $dims(H^{\pm})$ are instantiated according to $\mathcal{T}$. Similarly, $H^+[\mathcal{T}]$ is an upper bound. We also define:

$$\alpha(H^{\pm}) = max_{\mathcal{T}} \frac{H^+[\mathcal{T}]}{H^-[\mathcal{T}]}; \delta(H^{\pm}, \mathcal{T}) = 1 - \frac{H^-[\mathcal{T}]}{H^+[\mathcal{T}]}; \delta(H^{\pm}) = max_{\mathcal{T}} \delta(H^{\pm}, \mathcal{T}) \tag{7.3}$$

α is the standard *approximation ratio* known from approximation theory. $\delta(H^{\pm})$ is the maximal distance from the optimum (in percent) of any solution that will be implemented during the *VALUE* propagation. $\delta(H^{\pm})$ close to 0 or $\alpha(H^{\pm})$ close to 1 are equivalent, and guarantee solutions closer to the optimum.

If $H^{\pm}$ contains equal lower and upper bounds (as it happens in exact computation), it is easy to see from equation 7.3 that $\delta(H^{\pm}) = 0$, and A-DPOP reduces to DPOP.

So far we have described A-DPOP such that when the k bound is exceeded, then some dimensions are forcibly removed by approximate projections. Notice that it is in principle possible to compute and send *several* pairs of lower dimensionality upper/lower bound messages, each computed on a different subset of dimensions, in the spirit of the minibucket scheme[49].

7.2.2 *VALUE* propagation

As in DPOP, the *VALUE* phase proceeds top-down from the root after the *UTIL* phase. Upon receipt of the *VALUE* message from its parent, each node is able to pick the optimal value for itself according to one of the strategies from Equations 7.4 or 7.5.

Equation 7.4 selects as the optimal value the one with the minimal δ. We call this a δ-*strategy*. Notice that this will not necessarily produce the best assignment, since there may be another value that has a higher upper bound, but a worse δ. However, it offers the best *guaranteed* solution quality.

Equation 7.5 selects as the optimal value the one with the highest upper bound, even though it may not necessarily provide the best guarantees on solution quality (in case its lower bound is low). We call this *optimistic strategy*.

$$v_i^* = argmin_{v_i^j} \left(\delta \left(JOIN_i^{P_i \pm}, < agent\_view, X_i = v_i^j > \right) \right) \tag{7.4}$$

$$v_i^* = \left(JOIN_i^{P_i +} [agent\_view] \right) \perp_{X_i} \tag{7.5}$$

The algorithm terminates when all nodes have received *VALUE* messages and have assigned values to their variables.

7.2.3 A-DPOP complexity

As DPOP, A-DPOP produces a linear number of messages: $2 \times m$ DFS messages (m is the number of edges) and $n - 1$ UTIL and VALUE messages (n is the number of nodes). A-DPOP's complexity lies in the size of the *UTIL* messages (the *VALUE* messages have linear size):

Theorem 6 (A-DPOP complexity) *The largest UTIL message in A-DPOP is space-exponential in k or in the width induced by the DFS ordering used, whichever is smaller.*

PROOF. If the bound k is imposed and smaller than the width, no message larger than $O(exp(k))$ is produced (see Section 7.2.1). Then, complexity is exponential in this bound.

The worst case is when the exact solution is required ($k = \infty$, or $\delta = 0$). In this case, no dimensions can be dropped out of the *UTIL* messages, and *A-DPOP* reduces to *DPOP*, which is exponential in the induced width of the DFS used. □

7.2.4 Tradeoff solution quality vs computational effort and memory

It is easy to see that in case the parameter k is at least as big as the induced width of the problem, then all computations are exact, and the algorithm finds the optimal solution.

If not, then we have no choice but to use approximations: whenever a *UTIL* message exceeds the maximal dimensionality, approximate projections need to be applied. Optimality is thus lost, and we obtain an approximately optimal solution, and upper bounds on the distance from this solution to the true optimum.

Notice that approximate projections are applied only in high-width areas of the problem; for all the rest of the problem, where the dimensionality does not exceed k, optimal partial solutions are still found.

Another parameter that we can tune is the maximal error bound. This parameter enforces at each node an upper bound on the distance from the implemented solution to the true optimal solution for this node and its subtree. In case the deviation of the outgoing message is bigger than this bound, then we renounce a number of approximate projections until the bound is observed.

These two parameters are obviously conflicting. In case one cannot satisfy both of them, one needs to settle for the classical trade-off: accuracy vs. complexity. If optimality is the main concern, then one can specify e.g. $\delta = 10\%$, and no k. This would have as an effect that as many dimensions as needed would be used in order to guarantee that the obtained solution is within 10% of the optimum. Notice that this does not necessarily mean that the maximal number of dimensions will actually be used; depending on the valuation structure of the problem, one or two dimensions could very well be enough.

Conversely, if computation/network usage is the main concern, then one can specify e.g. $k = 2$ and no δ. In this case, the largest message would have 2 dimensions, and we would obtain the best solution available for this much computation, together with an upper bound on its distance from the true optimum. If this distance is good enough, then the algorithm returns this solution. Otherwise, we can re-run the algorithm with an increased k. Notice that in this case, we can reuse a lot of the previous work: one needs to re-run the propagation *only* in those areas of the problem where the maximal dimension bound was exceeded.

7.2.5 AnyPOP - an anytime algorithm for large optimization problems

In large, distributed constraint networks, it may take a long time until these propagations complete. In the following, we develop a way to decide quickly, *locally*, the value of each variable, based on a *limited* number of *UTIL/VALUE* messages from the neighbors. As time goes by, and the propagation spreads out, and more and more *UTIL/VALUE* messages come from the neighbors, we refine these

decisions. As opposed to a local search method, we obtain *guarantees* on the quality of the solution, even before allowing the propagations to complete. There are obvious advantages to this approach: one can quickly start with a reasonably good solution, and refine it as time goes by.

The intuition is simple: the value taken by any node X_i can have an influence on the rest of the problem only through the constraints between X_i and its direct neighbors. *UTIL* messages received by X_i already sum up its influence on the sending subtree. Thus, based on the set of *UTIL* messages X_i already received, and on the valuation structure of the constraints between X_i and its neighbors that did not already send *UTIL* messages, X_i can decide with a certain error bound what is the effect of each one of its values on the rest of the problem.

In some cases, when these error bounds are sufficiently low, X_i can decide on an assignment for itself even before receiving all of its *UTIL/VALUE* messages. In such a case, one can simply start the *VALUE* propagation phase immediately, without waiting for the rest of the *UTIL/VALUE* messages to come.

Let us first define

Definition 29 (Pseudoneighbor set PN_i^j) *The **pseudoneighbor set** PN_i^j is the set of pseudo-neighbors (pseudoparents or pseudochildren) of agent X_i that are reachable through its tree-neighbor X_j. e.g.: $PN_0^2 = \{X_{11}\}$, $PN_2^5 = \{X_{12}\}$, $PN_2^6 = \emptyset$.*

It is possible for an agent X_i to determine which is the tree-path associated with each one if its back-edges by comparing the suffix/prefix of the root-paths of its neighbors with their id's. Based on this, it is easy for X_i to determine PN_i^j for each neighbor X_j.

In the following subsections, we introduce the idea of *dominant values*, and present the *AnyPOP* algorithm (see algorithm 15) which makes use of them.

7.2.5.1 Dominant values

We present three increasingly weak kinds of dominant values.

Definition 30 (Statical dominance) *A value v_i^* of a variable X_i is a **statically dominant value** for X_i if v_i^* is the optimal value for X_i, no matter what values will X_i's neighbors take. Formally, v_i^* must always be $argmax_{X_i}\left(\bigoplus_{X_j \in Ngh(X_i)} R_j^i\right)$. If such a value is found, it is clear that X_i can already start the VALUE propagation, without waiting for any other message.*

7.2.5.2 Propagation dynamics

At any particular time t, we assume that a set of X_i's neighbors already sent X_i their *UTIL* messages. Let $Sent(X_i)$ be this set. According to Definition 29, each neighbor X_j of X_i, $X_j \in Sent(X_i)$ has an associated set PN_i^j. Any node $X_k \in PN_i^j$ (X_k is a pseudoneighbor of X_i), can reach X_i only through X_j. X_k does not directly send any *UTIL* message to X_i, but the relation R_i^k has already been taken into account in the message $UTIL_j^i$. This means that X_i can ignore the relation R_i^k, and consider X_k like it already sent an *UTIL* message.

Definition 31 (Extended sent set) *We define for a variable X_i the **extended sent set** as the set of tree neighbors of X_i which have already sent their UTIL messages, plus the pseudoneighbors of X_i which are reachable from X_i through these tree neighbors. Formally, $ExtSent(X_i) = Sent(X_i) \cup \{PN_i^j | X_j \in Sent(X_i)\}$*

Definition 32 (Dynamic join) *For a variable X_i we define the **dynamic join** $JOIN_i(t)$ as the join of the UTIL messages that have arrived, and of the relations with the neighbors that are not part of the extended sent set.*

$$JOIN_i(t) = \left(\bigoplus_{X_j \in \{Ngh(X_i) \setminus ExtSent(X_i)\}} R_j^i \oplus \bigoplus_{X_k \in Sent(X_i)} UTIL_k^i \right) \tag{7.6}$$

This dynamic join is a means to factor at any time the influence of X_i over the rest of the problem. $JOIN_i(t)$ takes into account utility information which is either explicit (*UTIL* messages from $Sent(X_i)$), implicit (the contribution of the relations with the pseudoneighbors from $ExtSent$ which is encapsulated in the received *UTIL* messages), or not decided (the relations with the neighbors which have not sent anything yet).

This dynamic join evolves with time: as more and more *UTIL* messages arrived, they replace the relations R_j^i in Equation 7.6, and the join has less and less dimensions.

Definition 33 (Dynamic dominance) *A value v_i^* of X_i is a **dynamically dominant value** for X_i if v_i^* is the optimal value of X_i for any values of X_i's neighbors except those in $ExtSent(X_i)$.*

Formally, if $agent\_view$ records the VALUE messages which were already received, and $JOIN_i(t)$ is the current dynamic join, a value v_i^* is dynamically dominant if v_i^* is always $argmax_{X_i}(JOIN_i(t)[agent\_view])$.

Notes: once such a value is determined for a variable, it cannot be changed by any incoming *UTIL* message. A statically dominant value is simply a dynamically dominant value computed before receiving any *UTIL* message.

7.2.5.3 Dynamically δ-dominant values

The two previous categories of dominance were exact: once found, a dominant value is certain to be the optimal value. We now present an approximative dominance: dominant values that allow for an error margin. They are computed in a very similar way with Equation 7.4:

$$v_i^{\delta *}(t) = argmin_{v_i^j} \delta \left(JOIN_i(t)^{\pm}, \langle agent\_view, X_i = v_i^j \rangle \right) \tag{7.7}$$

The value $v_i^{\delta *}(t)$ computed like in Equation 7.7 has the smallest guaranteed distance to the optimal solution, given the currently available information. It is obvious that as time progresses and more and more *UTIL/VALUE* messages arrive, the bounds become tighter and tighter, thus offering the possibility for increasingly accurate decisions.

If $\delta(t, v_i^{\delta *})$ is *small enough*, then we say that $v_i^{\delta *}(t)$ is a *dynamically δ-dominant value*, and we can safely assign it to X_i and start the *VALUE* propagation from X_i.

AnyPOP also exhibits some built-in fault tolerance. If messages are lost, there is a graceful degradation of performance: the δs will not be updated anymore, and in case that would have meant changing a current assignment, solution quality degrades. However, the algorithm still provides the best solution it can infer based on the information that *was* sent/received successfully.

7.2.6 Iterative A-DPOP for anytime behaviour

Another alternative for anytime solving is obtained by a straightforward iterative execution of A-DPOP with increasing bounds k, as described in Algorithm 16. Iterative A-DPOP starts with low values for k, which means that the UTIL messages sent are small, and can be quickly computed and sent over the network. This means that a (relatively) good solution can be obtained very fast. As time goes by, executions of A-DPOP(k) proceed, with increasing values of k, which mean that the approximate UTIL messages get larger and *and more accurate*, offering better bounds and better solutions. Like this we simulate an anytime behaviour with A-DPOP.

Remark 9 (Iterative A-DPOP can reuse computation between iterations.) *Notice that once the threshold k exceeds the size of a node X_i's separator, and of all descendants of X_i, the UTIL message computed and sent by X_i is* exact *(contains no approximations anymore). Afterwards, for subsequent executions of A-DPOP with larger values for k, X_i's parent P_i can simply reuse the $UTIL_i^{P_i}$ message it has previously received from X_i. Like this, X_i and its whole subtree have no more computation or message passing to do until the end of the algorithm. This effectively means that Iterative A-DPOP explores easy (low width) parts of the problem very fast in the beginning, and then most of the work is*

Algorithm 15 *AnyPOP - Anytime approximate Distributed Pseudotree Optimization*

AnyPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$, k, δ): each agent X_i does:

***UTIL* propagation protocol**

get all new *UTIL* messages (X_k, $UTIL_k^i$)
build $JOIN_i(t)$ as in Equation 7.6
if X_i *is root* **then** start *VALUE* propagation
else
  compute $\delta(t, v_i^j(t)), \forall v_i^j \in dom(X_i)$, and let $v_i^*(t) = argmin_{v_i^j}\left(\delta(t, v_i^j(t))\right)$
  if $\delta(t, v_i^*(t)) < \delta$ **then** start *VALUE* propagation
  if $|dims(JOIN_i^{P_i})| > k$ **then**
    select $S \subset dims(JOIN_i^{P_i})$ to be eliminated
    $UTIL_{X_i}^{P_i} = JOIN_i^{P_i} \perp_{S \cup \{X_i\}}^{\pm}$
  else $UTIL_{X_i}^{P_i} = JOIN_i^{P_i} \perp_{X_i}$
  Send $UTIL_{X_i}^{P_i}$ to P_i

***VALUE* propagation protocol**

get and store in $agent\_view$ all *VALUE* messages ($X_k \leftarrow v_k^*$)
recompute $\delta(t, v_i^j(t)), \forall v_i^j \in dom(X_i)$, and let $v_i^*(t) = argmin_{v_i^j}\left(\delta(t, v_i^j(t))\right)$
Send *VALUE*$(X_i \leftarrow v_i^*)$ to all C_i and PC_i

concentrated in the difficult parts of the problem.

Algorithm 16 *Iterative A-DPOP: Anytime based on iterative A-DPOP*

Iterative A-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$):

Construct DFS tree using Algorithm 3
run Algorithm 9; each $X_i \in \mathcal{X}$ knows Sep_i
$w = argmax_{X_i}|Sep_i|$ (the induced width)
for $k = 1 \ldots w$ **do**
  run A-DPOP(k) as follows:
  if $|Sep_i| < k$ *and* $UTIL_i^{P_i}$ *in A-DPOP($k-1$) was exact* **then**
    X_i *reuses* its UTIL message from A-DPOP($k-1$) in A-DPOP(k).
  set temporary solution according to A-DPOP(k)

7.2.7 Experimental evaluation

Our experiments were performed in the distributed meeting scheduling scenario described in [127] and in Section 2.3.1. In this context, the experiments were ran with an especially difficult problem containing 70 agents, 140 variables and 204 binary constraints. The induced width is 7, meaning that the largest message holds over two million values. We executed A-DPOP with increasing bounds on the

k	Max δ /msg %	Avg δ /msg %	δ /overall %	Total UTIL payload	Max msg size	Utility
1	44.83	13.55	2.90	2104	16	2278
2	36.00	4.54	2.69	10032	128	2283
3	17.14	1.27	2.43	39600	1024	2289
4	13.11	0.57	0.81	130912	8192	2327
5	10.00	0.19	0.43	498200	65536	2336
6	1.36	0.04	0.30	1808904	524288	2339
7	0.00	0.00	0.00	3614064	2097152	2346

Table 7.3: Max. dimensions vs. solution accuracy: problem with 140 vars, 204 constraints,width=7

Snapshot #	Max δ /var %	Avg δ /var %	Utility	δ /overall %	Assig changes
1	94.44	80.77	1555	33.72	0
2	66.07	16.7	1625	30.73	99
3	42.42	3.92	2036	13.21	73
4	13.51	1	2254	3.92	19
5	13.51	0.94	2289	2.43	1

Table 7.4: *AnyPOP* dynamic evolution: problem with 140 vars, 204 constraints,width=7

maximal dimensionality (k). We present in Table 7.3 the results in this order: maximal dimensionality, maximal δ for all *UTIL* messages (as in equation 7.3), the average δ per message, the distance of the approximate solution to the true optimum, the total amount of *UTIL* information transmitted (computed as the sum of the sizes of the individual *UTIL* messages), the maximal message size, and the utility of the solutions found. The accuracy of the solutions increases with the increase of k, culminating with the optimal value for $k = 7$, in which case A-DPOP(7) is equivalent to DPOP. However, there is also a dramatic increase in computation effort and network load. If we compare the first and the last lines of the table, we see that we can achieve a solution which is within 3% of the optimum with 3 orders of magnitude less effort (2k values sent over the network v.s. 3M). Therefore, if absolute optimality is not required, it might actually pay off to settle for a suboptimal solution obtained with much less effort.

We performed the second test with the same difficult instance from the previous test. This time, we wanted to test simultaneously both the anytime performance of *AnyPOP*, and its ability to deal with low resources. Therefore, we imposed $k = 3$, and started AnyPOP. We took a number of 5 snapshots of the assignments of the variables during the execution. The first snapshot is taken immediately after the initial δs are computed, *before sending/receiving any message*. Subsequent snapshots are taken after

each node has received another message. The last snapshot is taken after all messages are sent/received. The assignments discovered by each of the snapshots are used to compute the overall utility. We notice a steady progress of the algorithm towards a solution, culminating with the best solution found by *A-DPOP* on the same test problem, with the same bound $k = 3$. At the same time, there is a steady decrease of the error bounds, and of the assignment changes from one snapshot to the next.

7.2.8 Summary

We propose in this chapter an approximate algorithm for distributed optimization, which allows the desired tradeoff between solution quality and computational complexity. The algorithms can be extended with heuristics for selecting "intelligently" the dimensions to be dropped out when exceeding maximal message size. In the second part of the chapter we present an anytime version of the first algorithm, which provides increasingly accurate solutions while the propagation is still in progress. This makes it suitable for very large, distributed problems, where propagations may take a long time to complete. The anytime algorithm also exhibits some built-in fault-tolerance features, by graceful degradation of performance upon message loss. Experimental results show that these algorithms are a viable approach to real world, loose, possibly unbounded optimization problems.

Chapter 8

PC-DPOP: Tradeoffs between Memory/Message Size and Centralization

"Congrego et impera."
— Anonymous

In this chapter we discuss the idea of trading full decentralization for computational and communication efficiency. We introduce PC-DPOP, a new hybrid algorithm that is controlled by a parameter k which upper bounds the size of the largest message, and the amount of available memory. PC-DPOP(k) operates in the framework of Section 6.2 for detecting high-width clusters, where it is not possible to perform full inference as in DPOP. Such clusters are centralized in the root *of the cluster, and solved by the root in a centralized fashion, using an algorithm of its choice. Communication requirements over any link in the network are limited thus to $exp(k)$. The linear number of messages is preserved.*

In high width clusters, PC-DPOP offers a tradeoff between the fully decentralized solving process of DPOP for polynomial memory and message size. The overall behavior of PC-DPOP(k) is as follows: if w is the induced width of the problem given by the chosen DFS ordering, depending on the value chosen for k, we have 3 cases:

Fully decentralized algorithms for DCOP like DPOP or ADOPT often require excessive amounts of communication when applied to complex problems. Mailler and Lesser have introduced APO (Asynchronous Partial Overlay)[128], an algorithm which uses a strategy of partial centralization to mitigate this problem. While such a tradeoff is probably unfeasible in competitive settings where the agents

are non-cooperative (see the discussion from Section 11.6), in settings where the agents are perfectly cooperative, this approach can offer communication and computation savings.

In this chapter we introduce PC-DPOP, a new hybrid algorithm that is controlled by a parameter k which upper bounds the size of the largest message, and the amount of available memory. PC-DPOP(k) operates in the framework of Section 6.2 for detecting high-width clusters, where it is not possible to perform full inference as in DPOP because the memory requirements would exceed the bound imposed by k. In low-width areas, PC-DPOP proceeds as normal DPOP, using a linear number of messages and memory at most $O(exp(k))$. Clusters of high width are detected as in Section 6.2.1, and centralized in the *root* of the cluster. The cluster root then solves the subproblem in a centralized fashion, using an algorithm of its choice. Communication requirements over any link in the network are limited thus to $O(d^k)$. The linear number of messages is preserved.

Therefore, in these high width clusters, PC-DPOP offers a tradeoff between the fully decentralized solving process of DPOP for polynomial memory and message size. The overall behavior of PC-DPOP(k) is as follows: if w is the induced width of the problem given by the chosen DFS ordering, depending on the value chosen for k, we have 3 cases:

1. If $k = 1$, only linear size messages and memory are used.

2. If $k < w$, $PC-DPOP(k)$ performs full inference in areas of width lower than k, and centralization in areas of width higher than k. Memory and communication requirements are $O(exp(k))$.

3. If $k \geq w$, full inference is done throughout the problem; PC-DPOP(k) is then equivalent with DPOP (i.e. full inference everywhere). Memory requirements are $O(exp(w))$.

Partial results within each cluster are cached **(**8, 42, 132]) by the respective cluster root and then integrated as messages into the overall DPOP-type propagation. This avoids the need for any recomputation during the final VALUE propagation phase.

Compared to OptAPO, PC-DPOP provides better control over what parts of the problem are centralized and allows this centralization to be optimal with respect to the chosen communication structure. PC-DPOP also allows for a priori, exact predictions about privacy loss, communication, memory and computational requirements on all nodes and links in the network. We also report strong efficiency gains over OptAPO in experiments on three problem domains.

The rest of this section is structured as follows: Section 8.1 introduces the PC-DPOP hybrid algorithm, which is evaluated empirically in Section 8.2. Section 8.3 relates PC-DPOP to existing work. Section 8.4 briefly discusses privacy, and Section 8.5 concludes.

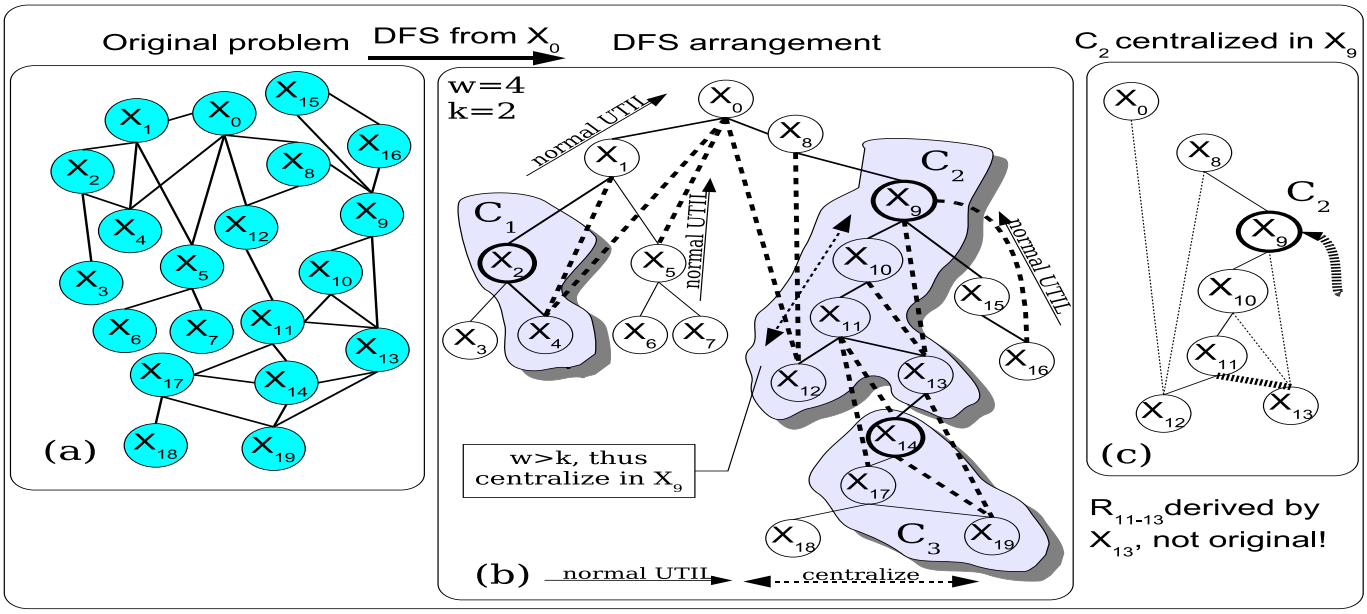


Figure 8.1: *A problem graph (a) and a DFS tree (b). In low-width areas the normal UTIL propagation is performed. In high width areas (shaded clusters C_1, C_2 and C_3 in (b)) bounded UTIL propagation is used. All messages are of size at most d^k. Clusters are centralized and solved in the cluster roots (the bold nodes X_2, X_9, X_{14}).*

8.1 *PC-DPOP(k)* - partial centralization hybrid

To overcome the space problems of DPOP, we introduce the control parameter k that bounds the message dimensionality. This parameter should be chosen s.t. the available memory at each node and the capacity of its link with its parent is greater than d^k, where d is the maximal domain size.

As with DPOP, PC-DPOP also has 3 phases: a *DFS construction* phase, an *UTIL* phase, and a *VALUE* phase. The DFS construction is simply done using Algorithm 3. Subsequently, on the established DFS structure, we run the LABEL-DFS algorithm from Section 6.2.1 (Algorithm 9). This algorithm identifies clusters of high width and labels the nodes as either *normal node*, *cluster-node* or *cluster-root node*. The subsequent UTIL phase assumes this labeling is in place.

8.1.1 PC-DPOP - *UTIL* Phase

This phase is an adaptation of the UTIL phase from DPOP. It proceeds as in DPOP for normal nodes, and reverts to partial centralization for cluster nodes (i.e. nodes whose separator size exceeds k):

1. the *UTIL* propagation starts bottom-up and proceeds exactly like in DPOP for normal nodes.
2. cluster nodes perform centralization (see Section 8.1.1.1): a cluster node does not compute its

UTIL message like in DPOP, but sends to its parent a *Relation* message that contains the set of relations (arity at most k) that the node would have used as an input for computing the *UTIL* message.

3. Upon receiving such a *Relation* message, a node X_i does:

   - If X_i is a cluster root, it reconstructs the subproblem from the incoming *Relation* messages and then solves it (see Section 8.1.1.3). Then it continues the *UTIL* propagation as in DPOP. Later on, during the VALUE phase, when X_i receives the *VALUE* message from its parent, it retrieves the solution from its local cache and informs nodes in the cluster of their optimal values via VALUE messages.
   - If X_i is a cluster node, it passes on to its parent all the relevant relations (the ones received from its children and its own), that it would otherwise use to compute its *UTIL* message. For details, see Section 8.1.1.1.

8.1.1.1 PC-DPOP - Centralization

Centralization occurs in high-width clusters such as C_1, C_2, C_3 in Figure 8.1. It is initiated by cluster nodes, since they cannot compute and send their *UTIL* messages because that would exceed the dimensionality limit imposed by k. Every cluster node packages together into a *Relation* message the union of the relations and UTIL messages received from children, and its own relations with its parent/pseudoparents. The resulting *Relation* message is sent to the parent, as in normal DPOP.

On one hand, this ensures the dimensionality limit k is observed, as no relation with arity larger than k is produced or sent over the network. On the other hand, this allows the cluster root to reconstruct the subproblem that has to be centralized, and enable the use of structure sensitive algorithms like DPOP, AOBB, etc.

Alternatively, to save bandwidth, avoid overload on cluster root nodes, and also improve privacy (see Section 8.4), a node can selectively join subsets of its outgoing *Relation* message, s.t. the dimensionality of each of the resulting relations is less than k. The resulting set of relations is then packaged as a *Relation* message, and sent to the parent. This happens as follows:

1. node X_i receives all *UTIL/Relation* messages from its children, if any
2. X_i forms the union U_i of all relations in the *UTIL/Relation* messages and the relations it has with its parent and pseudoparents
3. X_i matches pairs of relations in U_i s.t. by joining them the resulting relation will have k dimensions or less (the dimensionality of the resulting relation is the union of the dimensions of the inputs). If the join was successful, remove both inputs from U_i, and add the result instead. Try until no more joins are possible between relations in U_i. This process is linear in the size of U_i.

4. The resulting U_i set is sent to X_i's parent in a *Relation* message

This process proceeds bottom-up until a cluster root node X_r is reached. X_r then reconstructs the subproblem from its *Relation* messages, and solves it (see next Section).

The result is that in high-width clusters, the algorithm reverts to partial centralization, by having nodes send to their parents not high dimensional *UTIL* messages, but lower arity (aggregated) *inputs* that could be used to generate those *UTIL* messages.

8.1.1.2 Subproblem reconstruction

Let us assume a cluster root node X_i has received a set of relations R_{C_i} from its children. Each relation $r_i \in R_{C_i}$ is defined over a set of variables: $scope(r_i)$. X_i reconstructs the subproblem it has received as follows:

1. X_i creates an internal copy of all the nodes found in the scopes of the relations received.
2. X_i creates a hyper edge for each relation $r_i \in R_{C_i}$, which connects all variables in $scope(r_i)$.

It is interesting to note that this makes it possible for a cluster root to reconstruct the subproblem *while preserving structural information*. This is important because it enables the cluster root to use high-performance optimization algorithms that take advantage of problem structure, like for example[8, 42, 131, 132].

8.1.1.3 Solving centralized subproblems

The centralized solving occurs in the cluster root nodes. In the example of Figure 8.1, such a cluster is the shaded area containing $X_9, X_{10}, X_{11}, X_{12}, X_{13}$.

The root of the cluster (e.g. X_9) maintains a *cache table* that has as many locations as there are possible assignments for its separator (in this case $d^k = d^2$ locations). As a normal node in *DPOP*, the root also creates a table for the outgoing *UTIL* message, with as many dimensions as the size of the separator. Each location in the cache table directly corresponds to a location in the *UTIL* message that is associated with a certain instantiation of the separator. The cache table stores the best assignments of the variables in the centralized subproblem that correspond to each instantiation of the separator.

Then the process proceeds as follows:

- for each instantiation of Sep_i, the cluster root solves the corresponding centralized subproblem. The resulting utility and optimal solution are stored in the location of the *UTIL* message (cache table location, respectively) that correspond to this instantiation.

Algorithm 17 *PC-DPOP - partial centralization DPOP.*

PC-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$, k): each agent X_i does:

run ***LABEL-DFS*** protocol as in Algorithm 9 $\rightarrow$ X_i knows its label

***UTIL* propagation protocol**

wait for $UTIL/Relation$ messages from all children
if *label(X_i) = normal node* **then** compute $UTIL_i^{P_i}$ as in DPOP and send it to P_i
if *label(X_i) = cluster node* **then**
    Join subsets of incoming UTIL/Relation and relations with (p)parent with same dimension s.t. for each join, $dim(join) \leq k$
    package joins as $Relation_i$ and send to P_i
if *label(X_i) = cluster root node* **then**
    reconstruct subproblem from received relations
    solve subproblem for each $s \in \langle Sep_i \rangle$ and store utility in $UTIL_i^{P_i}$ and solution in local cache
    send $UTIL_i^{P_i}$ to P_i

***VALUE* propagation**(X_i gets $Sep_i \leftarrow Sep_i^*$ from P_i)
if X_i *is cluster root* **then**
    find in cache Sol^* that corresponds to Sep_i^*
    assign self according to Sol^*
    send Sol^* to nodes in my cluster via VALUE msgs
else continue VALUE phase as in DPOP

- when all Sep_i instantiations have been tried, the *UTIL* message for the parent contains the optimal utilities for each instantiation of the separator (exactly as in DPOP), and the cache table contains the corresponding solutions of the centralized subproblem that yield these optimal utilities.
- the cluster root sends its UTIL message to its parent, and the process continues just like in normal DPOP.

8.1.2 PC-DPOP - *VALUE* Phase

The labeling phase has determined the areas where bounded inference must be applied due to excessive width. We will describe in the following the processing to be done in these areas; outside of these, the original VALUE propagation from DPOP applies.

The VALUE message that the root X_i of a cluster receives from its parent contains the optimal assignment of all the variables in the separator Sep_i of X_i (and its cluster). Then X_i can simply retrieve from its cache table the optimal assignment corresponding to this particular instantiation of the separator. This assignment contains its own value, and the values of all the nodes in the cluster. X_i can thus inform all the nodes in the cluster what their optimal values are (via *VALUE* messages). Subsequently, the *VALUE* propagation proceeds as in DPOP.

8.1.3 PC-DPOP - Complexity

In low-width areas of the problem, PC-DPOP behaves exactly as DPOP: it generates a linear number of messages that are at most d^k in size. In areas where the width exceeds k, the clusters are formed.

Theorem 7 *$PC-DPOP(k)$ requires communication $O(exp(k))$. Memory requirements vary from $O(exp(k))$ to $O(exp(w))$ depending on the algorithm chosen for solving centralized subproblems (w is the width of the graph).*

PROOF. Section 8.1.1.1 shows that whenever the separator of a node is larger than k, that node is included in a cluster. It also shows that within a cluster, *UTIL* messages with more than k dimensions are never computed or stored; their input components are sent out instead. It can be shown recursively that these components have always less than k dimensions, which proves the first part of the claim.

Assuming that $w > k$, memory requirements are at least $O(exp(k))$. This can easily be seen in the roots of the clusters: they have to store the *UTIL* messages and the cache tables, both of which are $O(exp(Sep = k))$.

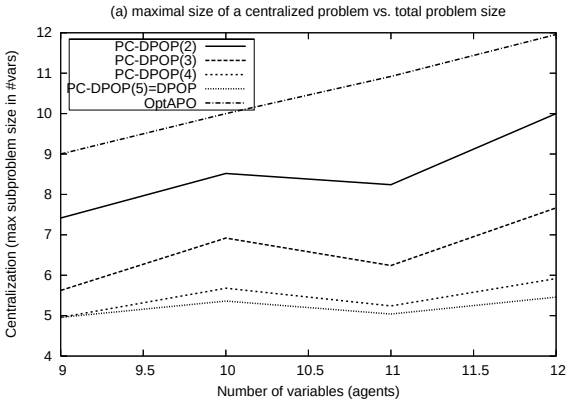
Within a cluster root, the least memory expensive algorithm would be a search algorithm (e.g. AOBB(1)) that uses linear memory. The exponential size of the cache table and UTIL message dominates this, so memory overall is $O(exp(k))$.

The most memory intensive option would be to use a centralized version of DPOP, that is proved to be exponential in the induced width of the subtree induced by the cluster. Overall, this means memory is exponential in the maximal width of any cluster, which is the overall induced width. □

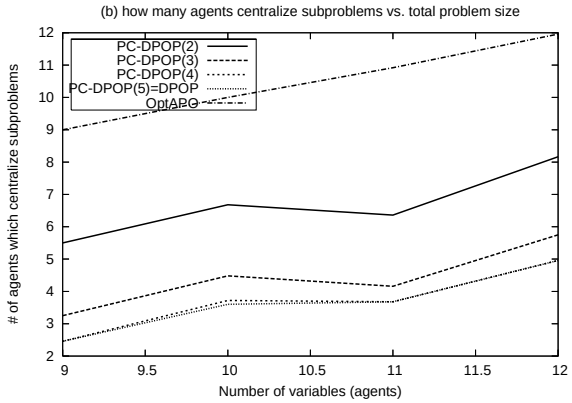
8.2 Experimental evaluation

We performed experiments on 3 different problem domains: graph coloring (GC, see Section 8.2.1), distributed sensor networks (DSN, see Section 8.2.2), and meeting scheduling (MS, see Section 8.2.3). For DSN and GC experiments we used the instances available online at[151], which are used in several other papers in the literature[127, 140].

Our versions of OptAPO and PC-DPOP used different centralized solvers, so in the interest of fairness, we did not compare their runtimes. Instead, we compared the effectiveness of the centralization protocols themselves, using 2 metrics: communication required , and amount of centralization. Overall, our results show that both OptAPO and PC-DPOP centralize more in dense problems; however, PC-DPOP's structure-based strategy performs much better.



(a) Maximal size of a centralized subproblem. In OptAPO there is always an agent which centralizes all the problem.



(b) Number of agents which centralize a subproblem. In OptAPO all agents centralize some subproblem.

Figure 8.2: PC-DPOP vs OptAPO: centralization in experiments on graph coloring.

8.2.1 Graph Coloring

The results from the GC experiments are shown in Figure 8.3 (communication requirements) and in Figure 8.2 (amount of centralization).

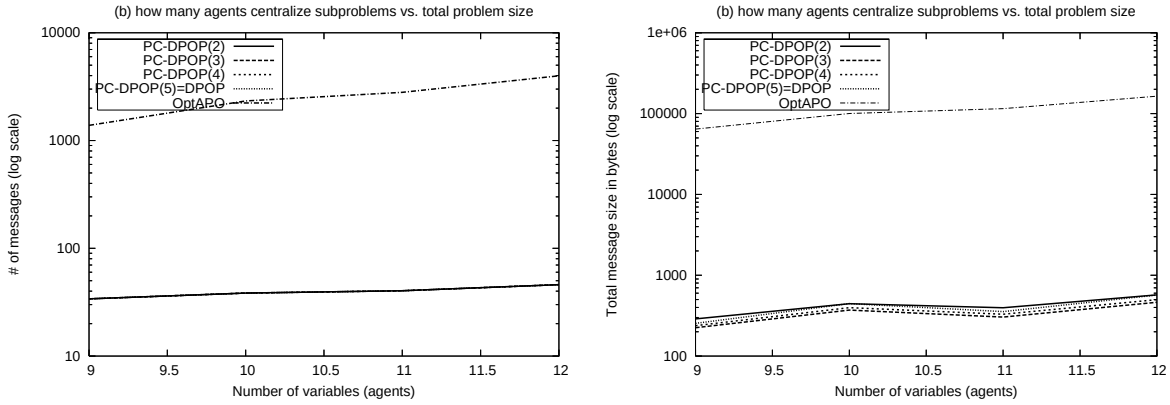
The bound k has to be at least as large as the maximal arity of the constraints in the problem; since these problems contain only binary constraints, we ran PC-DPOP(k) with k between 2 and the width of the problem. As expected, the larger the bound k, the less centralization occurs. However, message size and memory requirements increase.

8.2.2 Distributed Sensor Networks

The DSN instances are very sparse, and the induced width is 2, so $PC-DPOP(k \geq 2)$ always runs as DPOP: no centralization, message size is $d^2 = 25$. In contrast, in OptAPO almost all agents centralize some part of the problem. Additionally, in the larger instances some agents centralize up to half the problem.

8.2.3 Meeting scheduling

We generated a set of relatively large distributed meeting scheduling problems. The model is as in[127]. Briefly, an optimal schedule has to be found for a set of meetings between a set of agents. The problems were large: 10 to 100 agents, and 5 to 60 meetings, yielding large problems with 16 to 196 variables.



(a) All PC-DPOP variants use a linear # of messages.

(b) Total information exchange (bytes) is much lower for PC-DPOPs.

Figure 8.3: PC-DPOP vs OptAPO: message exchange in experiments on graph coloring.

The larger problems were also denser, therefore even more difficult (induced width from 2 to 5).

OptAPO managed to terminate successfully only on the smallest instances (16 variables), and timeout on all larger instances. We believe this is due to OptAPO's excessive centralization, which overloads its centralized solver. Indeed, OptAPO centralized almost all the problem in at least one node, consistent with[44].

In contrast, PC-DPOP managed to keep the centralized subproblems to a minimum, therefore successfully terminating on even the most difficult instances. *PC-DPOP(2)* (smallest memory usage) centralized at most 10% of the problem in a single node, and *PC-DPOP(4)* (maximal k) centralized at most 5% in a single node. *PC-DPOP(5)* is equivalent to DPOP on these problems (no centralization).

8.3 Related Work

The idea of partial centralization was first introduced by Mailler and Lesser in OptAPO[129]. See Section 3.3 for more details.

Tree clustering methods (e.g.[107]) have been proposed for time-space tradeoffs. PC-DPOP uses the concept loosely and in many parts of the problem transparently. Specifically, in areas where the width is low, there is no clustering involved, the agents following the regular DPOP protocols. In high-width areas, PC-DPOP creates clusters based on the context size of the outgoing UTIL messages and bounds the sizes of the clusters to a minimum using the specified separator size.

8.4 A Note on Privacy

Maheswaran et al.[126] show that in some settings and according to some metrics (complete) centralization is not worse (privacy-wise) than some distributed algorithms.

Even though the nodes in a cluster send relations to the cluster root, these relations may very well be the result of aggregations, and **not** the original relations.

Example 18 *For example, in Figure 8.1, X_{13} sends X_9 (via X_{11} and X_{10}) 3 relations: r_{13}^{11}, r_{13}^{10} and r_{13}^{9}. Notice that r_{13}^{11} that is sent to X_9 like this is not the real r_{13}^{11}, but the result of the aggregation resulting from the partial join performed with the UTIL message that X_{13} has received from X_{14}. Therefore, inferring true valuations may be impossible even in this scenario.*

8.5 Summary

We have presented an optimal, hybrid algorithm that uses a customizable message size and amount of memory. PC-DPOP allows for a priory, exact predictions about privacy loss, communication, memory and computational requirements on all nodes and links in the network.

The algorithm explores loose parts of the problem without any centralization (like DPOP), and only small, tightly connected clusters are centralized and solved by the respective cluster roots. This means that the privacy concerns associated with a centralized approach can be avoided in most parts of the problem. We will investigate more thoroughly the privacy loss of this approach in further work.

Experimental results show that PC-DPOP is particularly efficient for large optimization problems of low width. The intuition that dense problems tend to require more centralization is confirmed by experiments.

Part IV

Dynamics

Chapter 9

Dynamic Problem Solving with Self Stabilizing Algorithms

In this chapter we extend the discussion of distributed optimization algorithms to dynamically changing environments, like for example dynamic scheduling applications where tasks arrive and are executed continuously, or sensor networks where vehicles to be tracked move continuously.

These problems can be modeled as *dynamic CSPs*, and there is a wide body of research on this topic: [20, 52, 212, 216], to name just a few. We refer the interested reader to [211] for an excellent survey of various techniques that can be applied in this setting. However, the vast majority of these techniques operate in a centralized fashion: the dynamic changes in the environment are communicated to a central server, which then resolves the problem whenever necessary. In the following, we will present *distributed* algorithms for dynamic constraint reasoning; we focus on a class of algorithms called *self-stabilizing*.

Self stabilization in distributed systems is a concept introduced by Dijkstra in [57]. It is the ability of a system to respond to transient failures by eventually reaching a *stable state* where a *legitimacy predicate* is satisfied, and maintaining it afterwards. In the context of DCOP, we define the *legitimacy predicate* as "all variables are assigned to their values from the optimal solution of the DCOP".

Definition 34 (Self-stabilizing DCOP algorithm) *A DCOP algorithm is called* self-stabilizing *if it is able to always converge from any arbitrary initial configuration to a stable state where the legitimacy predicate is satisfied. This stable state corresponds to the optimal solution of the optimization problem, i.e. all variables in the problem are assigned their optimal values for the current problem configuration.*

Algorithms with this property are very well-suited to cope with error-prone distributed systems like distributed sensor networks, or with dynamic environments like control systems or distributed

scheduling, where convergence to stable states is ensured without user intervention. However, ensuring self-stabilization presents two major challenges. First, the algorithm must be able to deal with arbitrary state changes, like for example arbitrary changes in the problem topology (for example new agents coming in, or the network experiences temporary problems), in the valuations of the agents, or even in their internal data structures (for example as a result of temporary power outages). There is an obvious solution to this problem, namely simply restarting the optimization as soon as *any* change has happened in the problem. Nevertheless, this approach is most likely not practical, as it would raise another problem, namely the algorithm's ability to deal with successive changes that occur in a relatively fast sequence. The algorithm then must be fast enough in solving the changed problem, such that it is able to keep up with the changes.

These problems have so far mostly prevented self-stabilizing algorithms from addressing anything but relatively "low-level" tasks: leader election, spanning tree maintenance (e.g.[40]) and mutual exclusion. We will present in the following two notable exceptions: the earlier work of Collin, Dechter and Katz[39] for distributed self-stabilizing constraint satisfaction, and a self-stabilizing extension of the DPOP algorithm. We also note an attempt at self-stabilizing constraint optimization using a distributed, self-stabilizing version of branch and bound ([222]). This approach is not practical, however, since it may create an exponential number of agents, because they represent processes corresponding to subproblems.

9.1 Self-stabilizing AND/OR search

Collin, Dechter and Katz introduced in[39] the first self-stabilizing distributed constraint satisfaction algorithm. This algorithm also operates on a DFS tree.

In order to be able to guarantee self-stabilization, this algorithm uses a powerful principle: each agent executes continuously two parallel protocols: a DFS-contruction protocol, and a search protocol.

The DFS protocol they use (Collin and Dolev[40], also Dolev[59]) is guaranteed to eventually produce a valid DFS tree, provided no more changes happen in the problem structure.

The search protocol executes in parallel with the DFS generation protocol. It operates on the DFS tree that the first protocol produces. While this tree is not yet correctly established, the results are undefined. However, since the DFS protocol is guaranteed to eventually stabilize on a correct DFS, the search process is thus guaranteed to start operating on a correct DFS eventually. As the search process is also guaranteed to produce the correct solution in finite time given a correct DFS tree, it follows that the whole algorithm is self-stabilizing. For more details and a formal proof, see[39].

Using this *satisfaction* algorithm as a basis, one could in principle extend also *dAOBB* (Algorithm 2) for self-stabilizing *optimization*. Specifically, we use the same self stabilizing DFS protocol[40], interleaved with a self stabilizing version of the search protocol executed in *dAOBB*. The latter protocol can

be easily made self-stabilizing by having all agents cycle continuously through their values (forward search phase) and propagate cost bounds to their ancestors (backward bound propagation).

9.2 Self-stabilizing Dynamic Programming: S-DPOP

The self-stabilization principles from Collin, Dechter and Katz[39] can be extended straightforwardly to DPOP as well[165]. We propose a method that is composed of 3 concurrent self-stabilizing protocols:

1. self-stabilizing protocol for DFS tree generation: as in[40], its goal is to create and maintain (even upon faults/topology changes) a DFS tree maintained in a distributed fashion.
2. self-stabilizing protocol for propagation of utility messages: bottom-up utility propagation along the DFS tree, as in Section 4.1.2.
3. self-stabilizing protocol for propagation of value assignments: based on the utility information obtained in protocol 2, each agent picks its optimal value and informs its children (top-down along the DFS tree, as in Section 4.1.3).

The three protocols are initialized and then run concurrently. The resulting method, called *S-DPOP* is described in Algorithm 18.

Proposition 11 *Algorithm S-DPOP is self-stabilizing as specified in Definition 34.*

PROOF. We follow the same line of reasoning as in[39]. Specifically, S-DPOP is composed of the three self-stabilizing sub-protocols described previously. First, the DFS generation subprotocol is guaranteed to self-stabilize, and eventually produce a correct DFS[40]. Second, the UTIL propagation subprotocol is guaranteed to execute correctly after the DFS is correctly constructed, and self-stabilize after $n-1$ UTIL messages. Third, the VALUE propagation subprotocol is guaranteed to execute correctly after the UTIL subprotocol has provided all agents in the system with accurate UTIL information. Therefore, the whole S-DPOP protocol is guaranteed to self-stabilize. □

9.2.1 S-DPOP optimizations for fault-containment

In a dynamic setting, many different changes can occur in the optimization problem: valuations can change, variables and constraints can be removed or added, etc. We describe in the following several possible optimizations to S-DPOP which make it more responsive to changes by increasing the reusability of previous computation, and by limiting the propagation of new messages upon perturbations. In doing so, we touch upon aspects of *fault containment*[82], which means that *minor* changes can effectively be contained to confined areas in their vicinity.

Algorithm 18 *S-DPOP - Self-stabilizing DCOP algorithm.*

S-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): each agent X_i runs 3 subprotocols simultaneously:

Self-stabilizing DFS protocol: run continuously the protocol from[40]

1 at stabilization, X_i knows P_i, PP_i, C_i, PC_i

***UTIL* propagation protocol**: run continuously - wait for UTIL messages

2 **if** *received new UTIL msg (X_k, $UTIL^i_k$) OR P_i, PP_i, C_i, PC_i or R^k_i changed* **then**

3 recompute $UTIL^{P_i}_{X_i} = \left(\left(\bigoplus_{c \in C_i} UTIL^i_c \right) \oplus \left(\bigoplus_{c \in \{P_i \cup PP_i\}} R^c_i \right) \right) \perp_{X_i}$

4 Store $UTIL^{P_i}_{X_i}$ and send it to P_i

***VALUE* propagation protocol**: run continuously - wait for VALUE messages

5 **if** *received new VALUE msg (X_k, $v(X_k)$) OR changes in $UTIL^{P_i}_{X_i}$* **then**

6 $v^*_i \leftarrow argmax_{X_i} \left(UTIL^{P_i}_{X_i}[v(P_i), v(PP_i)] \right)$

7 Send $VALUE(X_i, v^*_i)$ to all C_i and PC_i

9.2.1.1 Fault-containment in the DFS construction

Changes in the DFS structure adversely affect the performance of S-DPOP, since some of the *UTIL* messages will have to be recomputed and retransmitted. Therefore, it is desirable to maintain as much as possible the current DFS tree upon a change, to be able to reuse most of the effort that was spent while solving the previous problem instance. After the new DFS is constructed, it is easy to decide which UTIL messages can be reused, by comparing the new DFS with the old one. All messages computed and sent in parts of the problem where the DFS was not affected can be reused.

We will describe in the following a number of simple modifications to the problem, and the corresponding changes they induce to the DFS tree.

Additions to the problem Adding a new variable X_i to the problem (and a new relation r^j_i to link it with an existing one, X_j): this is a trivial case. One has just to connect X_i as a child of X_j. X_i simply starts a propagation by sending X_j the projection $r^j_i \perp X_i$. In the worst case, this propagates to all the ancestors, up to the root. This implies in the worst case a number of UTIL messages that equals the number of ancestors of X_j, and the same amount of effort that was spent in the original propagation along this path. [1]

Adding a new relation/constraint between 2 existing agents, X_i and X_j. Depending on the relative position of X_i and X_j, we have 2 cases:

1. **X_i and X_j are ancestor-descendant** (they lie in the same branch of the DFS): this a simple

[1] For example, in Figure 9.1, if one adds a variable X_{14}, connected with a single constraint to X_{13}, then it becomes X_{13}'s child, and the DFS does not suffer any other modifications.

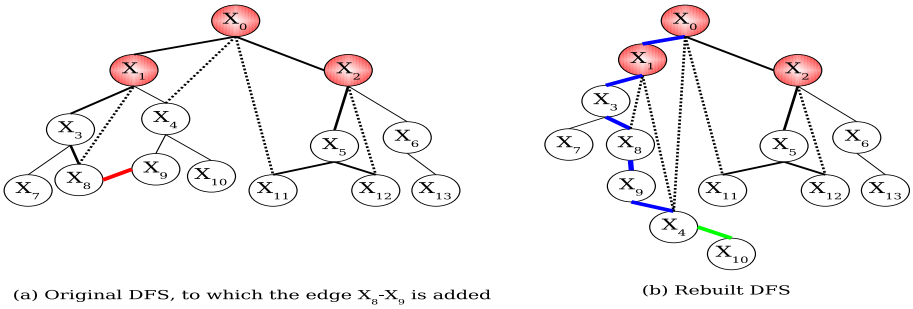


Figure 9.1: *Additions to a problem: the most difficult case is case 2 from Section 9.2.1.1 of adding an edge between siblings. Adding the red edge $X_8 - X_9$ disrupts the DFS from (a) to (b). In (b), the blue lines denote the messages that have to be recomputed in the worst case. Notice that $X_{10} - X_4$ (the green edge) does not change, so $UTIL_{10}^4$ does not require re-computation.*

case, we just need to designate the new edge as a back-edge. Assuming (without loss of generality) that X_i is the descendant, X_j becomes X_i's pseudoparent. The *UTIL* propagation needs to be restarted only from X_i, and to incorporate the newly added backedge. X_i can reuse all the messages it has previously received from its children. [2]

2. ***X_i* and *X_j* are siblings** (they lie in different branches of the DFS): adding such an edge violates the required property that agents in different branches of the DFS be disconnected. This implies that the DFS is no longer valid, and it has to be reconstructed. To maximize the similarity to the previous DFS arrangement (and therefore the reuse of UTIL messages), we propose a simple repair heuristic. Either one of X_i or X_j becomes a parent for the other one. Without loss of generality, let us assume that X_i becomes X_j's parent. Let X_k be the lowest common ancestor of X_i and X_j. The required changes concern only the agents on the tree-path from X_j to X_k: they all switch their parent-child roles, except for the immediate child of X_k, which becomes its pseudochild. All other agents are unaffected. [3]

Deletions from the problem Deleting a constraint: depending on the type of the edge, we have 2 cases:

1. **deleting a back-edge**: we simply remove the back-edge, and the lower agent involved in that back-edge restarts a UTIL propagation without including the dimension of its (former) pseudoparent. [4]

[2] For example, in Figure 9.1, if one adds an edge $X_9 - X_1$, then this edge simply becomes a back-edge, and X_9 becomes a pseudochild of X_1. The DFS does not suffer any other modifications.

[3] For example, in Figure 9.1, if one adds an edge $X_8 - X_9$, then X_8 becomes X_9's parent, and agents on the path from X_9 to X_1 switch roles: X_4 becomes X_9's child. Additionally, X_4 also becomes X_1's pseudochild. The DFS does not suffer any other modifications (e.g. X_{10} remains X_4's child).

[4] For example, in Figure 9.1, if one deletes the edge $X_8 - X_1$, then X_8 simply restarts the UTIL propagation with just

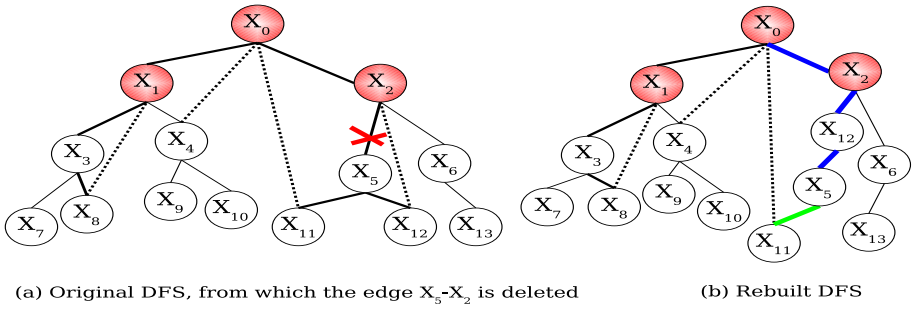


Figure 9.2: *Deletions from a problem: the most difficult case is case 2.b from Section 9.2.1.1 of deleting a tree edge that does not disconnect the problem. Deleting the red edge $X_5 - X_2$ disrupts the DFS from (a) to (b). In (b), the blue lines denote the messages that have to be recomputed in the worst case. Notice that $X_{11} - X_5$ (the green edge) does not change, so $UTIL_{11}^5$ does not require re-computation.*

2. **deleting a tree-edge**: let X_i and X_j be the two agents involved in it (X_i is X_j's parent). We have again two cases:

   (a) If $Sep_j = \{X_i\}$, and also $\forall X_k \in C_j, Sep_k = \{X_i\}$, then by removing the edge $X_i - X_j$ we have effectively disconnected the problem in two distinct parts: the subtree rooted at X_j, and the rest. X_j becomes a root now, so it can initiate a VALUE propagation based on the UTIL information it already has collected from the subtree. For the rest of the problem, X_i starts a new UTIL propagation by recomputing its UTIL message while disregarding the message previously sent from the (now disconnected) subtree of X_j. [5]

   (b) Otherwise, removing the edge $X_i - X_j$ does not disconnect the problem, but disrupts the tree, however. One needs to restart the DFS reconstruction from the highest agent in Sep_j. Let X_k be this agent. X_k restarts the DFS reconstruction by sending DFS messages to its children and pseudochildren. There is no point in sending these messages to its parent/pseudoparents, since they cannot be affected by the removal of the edge. This is so because X_k is the highest agent connected with X_j's subtree.

   The DFS reconstruction proceeds then as normal in the whole subtree rooted at X_k, which includes the area affected by the removal of the edge $X_i - X_j$. [6]

Note: All other complex changes can be decomposed into a sequence of simple changes like the ones described before. For example, deleting a variable and all its constraints amounts to deleting its

X_3 as a dimension.

[5] For example, in Figure 9.2, consider removing the edge $X_2 - X_6$.

[6] For example, in Figure 9.2, consider removing the edge $X_5 - X_2$. $Sep_5 = \{X_0, X_2\}$, so the highest agent in Sep_5 is X_0. The DFS reconstruction restarts thus from X_0, in its right-hand side subtree. The traversal proceeds as follows: $X_0 \rightarrow X_2 \rightarrow X_{12} \rightarrow X_5 \rightarrow X_{11} \rightarrow X_0$. At this point, the DFS reconstruction is complete, and the result is depicted in Figure 9.2(b). Notice the role changes: X_{12} is now X_2's child (not a pseudochild anymore) and X_5 and X_{12} have switched parent/child roles. The blue edges represent UTIL messages that have to be recomputed, while the green one ($X_{11} - X_5$) shows that the $UTIL_{11}^5$ message can be reused.

Algorithm 19 *Fault containment in SS-DPOP - limiting the spread of UTIL/VALUE propagations.*

***UTIL* propagation protocol**:
step 3.a: find v=min($UTIL_{X_i}^{P_i}$); subtract v from each cell in $UTIL_{X_i}^{P_i}$
step 3.b: **if** *new* $UTIL_{X_i}^{P_i}$ = *old* $UTIL_{X_i}^{P_i}$ **then** discard new $UTIL_{X_i}^{P_i}$

***VALUE* propagation protocol**:
step 6.a: **if** *new* v_i^* = *old* v_i^* **then** do not send VALUE message

constraints one by one, until it has a single one left (the last step is obvious). Adding a variable and several constraints amounts to adding a variable and a single constraint, and then adding constraints between existing variables.

9.2.1.2 Fault-containment in the *UTIL/VALUE* protocols

In S-DPOP, upon a perturbation all *UTIL* messages on the tree-path from the change to the root are recomputed and retransmitted; subsequently, *VALUE* messages circulate top-down throughout the problem. This is sometimes wasteful, since some of the faults have limited, localized effects, which need not propagate through the whole problem. We change *S-DPOP* (Algorithm 18) by adding three steps, presented in Algorithm 19

Steps 3.a and 3.b are designed to identify and cut irrelevant UTIL propagations, and step 6.a to cut irrelevant VALUE propagations.

Step 3.a *rescales* all *UTIL* matrices by subtracting from each element the lowest utility value present in that matrix. This is a sound operation for computing the optimal solution, because in DPOP the relative differences in utility are important, and not the absolute valuations: we just want to find the optimal solution, we do not care about its utility. [7] Step 3.b compares the newly computed UTIL message with the previous one; in case there are no differences, it is simply discarded. Thus, through rescaling and projections, the influences of a change in terms of utility variations diminish from one hop to the next, until the propagation stops altogether.

9.2.2 S-DPOP Protocol Extensions

Self stabilizing algorithms generally do not provide any guarantees about the way the system transits from a valid state to the next, upon perturbations. The following two sections show that in some circumstances, we can provide transitional guarantees via superstabilization and fast responses upon low impact changes.

[7] Intuitively, if an agent X_i has 3 values `[a,b,c]`, then receiving `[0,1,2]` as valuations for its values is the same as receiving `[10,11,12]`: it still means that value `c` yields 2 units of utility more than value `a`, and 1 unit of utility more than value `b`, and will thus be chosen as optimal.

9.2.2.1 Super-stabilization

Super-stabilization[59, 60] is a guarantee that a self-stabilizing protocol satisfies a *passage predicate* at all times, transitional states included. Formally,

Definition 35 (Superstabilization) *A protocol $\mathcal{P}$ is said to be* superstabilizing *with respect to a* passage predicate p *for a class of changes Λ if and only if $\mathcal{P}$ is self-stabilizing, and for every trajectory Φ beginning at a legitimate state and containing a single change of type Λ, the passage predicate p holds for every $\sigma \in \Phi$.*

Recall that for DCOPs, we defined the legitimacy predicate in Definition 34 as "all variables are assigned to their optimal values". For our purposes, we define the passage predicate $p =$"*the previous optimal assignment is maintained while the new one is recomputed, and the switch is made atomically*". We also define the class Λ as any changes in the problem which do not invalidate the current solution, i.e. they do not make it inconsistent: adding values to a variable, adding / removing / changing a relation, removing a constraint, and even adding a constraint, as long as it does not forbid (parts of) the current assignment (that would clearly invalidate the current assignment).

A super stabilizing algorithm with respect to predicate p and changes in class Λ as defined above, ensures that a consistent solution (i.e. the previous optimal solution, which has now possibly become suboptimal) is maintained at all times, even in transitory states. Superstabilization w.r.t this passage predicate p can be regarded as a safety property, weaker than the legitimacy predicate, but nevertheless useful: this guarantee of consistency can be important for example in control systems, where inconsistent assignments cannot be tolerated in transitory states where the algorithm searches the new best solution after a fault.

SS-DPOP (Algorithm 20) relies on additional assumptions to guarantee super stability: the agents have synchronized clocks, the messages are transmitted synchronously, and each agent knows (a) its level in the DFS tree and (b) the depth of the DFS tree (both can be made available by the DFS construction protocol). The algorithm works as S-DPOP: upon a fault, the agents start to recompute and resend the UTIL and VALUE messages. However, now all the agents must switch their values to their new optimum synchronously, in an *atomic step*, to avoid transitory inconsistent assignments. They synchronize by delaying the switch to the new value: assuming the transmission of a VALUE message takes a clock "tick", each agent delays switching its value for a number of ticks equal to the difference between the depth of the DFS and its own level in the DFS. This ensures that the switch is made at the moment when the last leaf agent has received the VALUE message from its parent, and can compute its own optimal value.

Proposition 12 *SS-DPOP is super stabilizing in the sense of Definition 35.*

Algorithm 20 *SS-DPOP - Super-stabilizing DCOP algorithm.*

SS-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): changes from S-DPOP (Algorithm 18)

***VALUE* propagation protocol**: run continuously - wait for VALUE messages

1 **if** *received new VALUE msg (X_k, $v(X_k)$) OR changes in $UTIL_{X_i}^{P_i}$* **then**

2 $v_i^{tmp} \leftarrow argmax_{X_i} \left(UTIL_{X_i}^{P_i}[v(P_i), v(PP_i)] \right)$

3 Send $VALUE(X_i, v_i^*)$ to all C_i and PC_i

4 wait for $depth - level$ clock ticks

5 assign $v_i^* = v_i^{tmp}$

PROOF. When a failure $\sigma \in \Lambda$ occurs, the agents preserve their current assignments. By definition of the class Λ, this ensures $p = true$. Agents then recompute and resend their UTIL messages. When the root (level 0 in the DFS) has received all updated messages, it decides for its new value, and sends VALUE messages to its children. It will then wait for `depth` clock ticks before it actually sets itself to this new value. We assume messages are delivered synchronously, thus they arrive in the following clock tick at the nodes on level 1, which send VALUE messages and wait for `depth-1` ticks, and so on. The VALUE propagation phase takes thus `depth` clock ticks, and at that time all nodes switch to their new optimal values in a synchronized manner. □

9.2.2.2 Fast response time upon low-impact faults

In dynamic systems, optimal decisions have to be made as quickly as possible. In some cases, we want to respond *immediately* to a perturbation by re-assigning the "touched" variable to its new optimal value, and then gradually re-assigning the neighboring ones to their new optimal values, until the whole system re-stabilizes.

Definition 36 (Low impact faults) *A low impact fault on a variable X_i is the addition of a constraint that further limits the available values for X_i, or changes the local utilities associated with some of its values.*

To be able to immediately assess *locally* the *global* effect of such a fault, each agent needs global utility information. To this end, we use the bidirectional utility propagation extension from Section 4.1.6. Then, each agent simply joins together all the UTIL messages received from its parent and children, and then projects out all other variables except itself. This gives the agent a global view of the whole problem, as it produces a utility vector that accurately describes what is the best utility achievable by the whole problem *for each one of the values of the agent in question*.

Algorithm 21 *LIF-S-DPOP - Dynamic DCOP algorithm (changes from S-DPOP)*

LIF-S-DPOP($\mathcal{X}$, $\mathcal{D}$, $\mathcal{R}$): changes from S-DPOP (Algorithm 18)

***UTIL* propagation protocol**: bidirectional version, as in Section 4.1.6

1 **if** *received new UTIL msg (X_k, $UTIL_k^i$) OR P_i, PP_i, C_i, PC_i or R_i^k changed* **then**

2 recompute $JOIN_{X_i} = \left(\left(\bigoplus_{c \in \{C_i \cup P_i\}} UTIL_c^i\right) \oplus R_i\right)$

3 $UTIL_{X_i}^{global} = JOIN_{X_i} \perp_{X_j \neq X_i}$

***VALUE* propagation protocol**: run continuously - wait for VALUE messages

4 **if** *changes in $UTIL_{X_i}^{global}$* **then**

5 $v_i^* \leftarrow argmax_{X_i}\left(UTIL_{X_i}^{global}\right)$

6 Send $VALUE(X_i, v_i^*)$ to all C_i and PC_i

Once this vector is available, dealing with a low-impact fault is easy: X_i simply has to join the new relation/constraint to the vector, and it finds out what is its best value in the new situation [8] . This later step requires no communication, and only a linear amount of computation.

The resulting algorithm is presented in Algorithm 21.

Proposition 13 *Algorithm 21 self-stabilizes in response to a low-impact fault in a time delay of n VALUE messages.*

PROOF. When a low-impact fault occurs at an agent X_i, X_i immediately finds out its new optimal value by joining the new relation/constraint describing the fault with the pre-computed $UTIL_{X_i}^{global}$ vector, and choosing the new best value. Afterwards, X_i announces its neighbors of the change by sending VALUE messages. When another agent X_j receives a new VALUE message, it simply retrieves its best response from its internal $JOIN_{X_j}$ message, and announces its own neighbors about the change, and so on. The whole propagation stops after at most n VALUE messages, i.e. in the worst case after all the agents in the problem change value. □

[8] This assumes that there are no other simultaneous changes in the problem.

Chapter 10

Solution stability in dynamically evolving optimization problems

In dynamic systems, changes occur all the time, and optimization is a continuous process. In some cases, it is required to decide on the values of at least a subset of the variables of the problem, and fix them to some desirable values. A simple example is a dynamic scheduling problem, where at some point one has to fix some tasks and start working on them, otherwise deadlines would not be kept.

The traditional dynamic CSP model[18, 20, 52, 212, 216] deals with dynamic environments by assuming that the CSP solver has to deal with a sequence of static CSPs. The solver solves each one of these CSPs, and finds the optimal solution at each step. In some settings, it is important to try to minimize the number of variable assignments that differ between successive solutions. For example, when a new task is given to a scheduler, it may be wasteful to re-schedule all the other schedules that were previously computed; it may be desirable to disrupt the existing schedule as little as possible. For this purpose, the objective of *solution stability* was introduced ([83, 216]), which states that solutions to successive problems should differ in as few variable assignments as possible.

We next extend the traditional dynamic CSP formalism along two dimensions. First, we introduce a more flexible mechanism to deal with environment dynamics (Section 10.1), and second, we introduce an effective mechanism for evaluating and maintaining solution stability for a problem that evolves with time (Section 10.2).

10.1 Commitment deadlines specified for individual variables

First, we introduce a new level of granularity as far as time is concerned. We do not treat the dynamically evolving CSP as a sequence of CSPs that have to be solved individually, but rather introduce the idea of per-variable *commitment deadline*. In this approach, upon defining the optimization problem,

the designer has the opportunity to specify *commitment deadlines* for each variable: deadlines until which a value must be assigned to the respective variable. This gives more flexibility, as each variable is treated individually, and the ones that do not have to be committed to any value do not interfere with maintaining solution stability.

We identify two kinds of commitments:

1. *Soft commitments* model contracts with penalties, and can be revised if the benefit extracted from the change outweighs its cost. The penalties associated with changing a variable assignment (a decision that has been made) are modeled with *stability constraints* (see definition 37 in Section 10.2).

2. *Hard commitments* model irreversible processes, and are impossible to undo (example: production of good *X* already started, and resource *Y* was already consumed). When a variable is hard-committed to a value, the variable can be removed from the problem.

10.2 Solution Stability as Minimal Cost of Change via Stability Constraints

Current approaches define solution stability in dynamic CSP with respect to the number of variable assignments that need to be changed in order to reach again a consistent state upon a change in the problem. There are two approaches to achieve this kind of stability. The first approach (e.g.[212]) is *reactive*: once a change occurs in the problem, one seeks the new solution which is closest to the previous one, thus requiring a minimal number of changes. The second approach (e.g.[27, 99, 216]) is *proactive*: when generating a solution in the first place, one tries to find robust solutions, which are likely to remain valid even upon changes in the problem, thus requiring little or no adjustment. [27] uses a probabilistic model that tries to predict what possible changes can happen in the future, and tries to generate solutions that are robust with respect to the predicted changes.

Our approach falls in the category of *reactive* approaches. We do not try to predict future changes, or to build robust solutions; rather, we simply optimize continuously and provide the optimal solution at all times. However, we break away from the traditional definition of solution stability by looking at the process from a *cost* perspective. We argue that the number of assignments that change is irrelevant; what matters is the total *cost* that is incurred by performing these changes, given the current assignments.

Therefore, we introduce *stability constraints* to allow for such changing costs to be explicitly modeled into the COP framework with *stability constraints*:

Definition 37 (Stability Constraint) *A **stability constraint** σ_i is a function $\sigma_i : dom(X_i) \times dom(X_i) \rightarrow \mathbb{R}$,*

s.t. $\sigma_i(v_i^j \rightarrow v_i^j) = 0$ *(it does not cost anything if* X_i *stays unchanged). The semantics of such a constraint is simple: if* X_i *is assigned to* v_i^1*, then* $\sigma_i(v_i^1 \rightarrow v_i^2)$ *denotes how much it costs to change* X_i*'s value to* v_i^2*.*

We define the distributed, continuous-time combinatorial optimization problem:

Definition 38 (DynDCOP) *Formally, a discrete* ***dynamic distributed constraint optimization problem*** *(DynDCOP) is a tuple* $< \mathcal{X}, \mathcal{D}, \mathcal{R}, \mathcal{S}, \mathcal{T} >$ *that extends the DCOP definition with:*

- $\mathcal{S} = \{\sigma_1, ..., \sigma_m\}$ *is a set of stability constraints*
- $\mathcal{T} = \{t_1, ..., t_m\}$ *is a set of commitment deadlines: times until the corresponding variable has to commit to a value. Deadlines can be specified for hard or soft commitments.*

Notice that this model of a DynDCOP is purely reactive: we do not assume any knowledge or model of future events. At each moment, we seek the current optimal solution to the problem, taking into account the costs incurred from revising previous commitments. Formally, we define the new optimal solution to a dynamic DCOP as follows:

$$\mathcal{X}^*_{new} = argmax_{\mathcal{X}} \left(\sum_{r_l \in \mathcal{R}} r_l(\mathcal{X}) - \sum_{\sigma_i \in \mathcal{S}} \sigma_i(\mathcal{X}^{old} \rightarrow \mathcal{X}) \right) \tag{10.1}$$

where the first sum is the utility of the new solution, and the second sum is the cost one has to pay for changing the current assignments to the new ones.

For uncommitted variables, the cost is 0: they can simply choose their new optimal values, without any cost. Hard-committed variables cannot change their values anymore (one can think of it as an infinite change cost).

Thus, what we need to optimize is the difference between the new utility and the cost associated with changing the soft-committed variables. Section 10.3 introduces RS-DPOP, an algorithm which implements this idea.

10.3 Algorithm RS-DPOP

This section introduces RS-DPOP, an extension of the self stabilizing algorithm S-DPOP. RS-DPOP implements the two extensions that we introduced to the DynDCOP framework: individual commitment deadlines, and implementation of solution stability as cost of changing committed assignments.

There are two changes from the original S-DPOP. First, we add a time monitor for each agent that handles the deadlines imposed on the commitment of its variable. Second, the *UTIL* and *VALUE*

propagations are changed as far as the committed variables are concerned. The *RS-DPOP* algorithm is described in Algorithm 22.

10.3.1 *UTIL* propagation

The *UTIL* propagation is essentially the same as in *S-DPOP*, with the exception of the committed variables.

Soft Commitments and Stability Constraints: suppose X_i has already soft-committed to v_i^*. If there are some changes in the problem, and X_i needs to resend its *UTIL* message to its parent, then it can recompute it by adding the cost of change to the current $JOIN$, followed by an optimal projection along its dimension: $UTIL_{X_i}^{P(i)} = \left(JOIN_i^{P(i)} \oplus \sigma_i[v_i^*]\right) \perp_{X_i}$.

For each tuple of variables in $\{dim(JOIN_i^j) \setminus X_i\}$, all the corresponding values from $JOIN_i^j$ (one for each value of X_i) are considered. The value corresponding to $X_i = v_i^*$ is not modified - no change, no cost. From all the other values corresponding to $X_i = v_i^k, k \neq *$ we substract "the cost of change": $\sigma_i(v_i^* \rightarrow v_i^k)$. We then choose the best value. Computing the *UTIL* messages like this ensures that the utility values sent by X_i are either computed by keeping the same value for X_i, or take into account the cost of change.

Hard commitments: When computing its *UTIL* messages, a hard-committed variable cannot change its value anymore, so instead of an optimal projection, a slice is used: if X_i was already assigned to v_i^*, then $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)}[X_i = v_i*]$

10.3.2 *VALUE* propagation

Now the optimization of the local value happens only if the variable is not hard-committed. If it is soft-committed, the cost of change is taken into account. Otherwise, the variable is "floating", and it can freely be changed to its new value.

Proposition 14 (RS-DPOP correctness) *Algorithm 22 is correct in the sense that it correctly finds the (instantaneous) optimal solution according to Definition 10.1.*

PROOF. Follows from the fact that the stability constraints are taken into account while computing the UTIL messages (step 4 in Algorithm 22) and while determining the new optimal assignments in the VALUE phase (step 7 in Algorithm 22). □

Algorithm 22 *RS-DPOP - Dynamic DCOP algorithm (changes from S-DPOP)*

$(\mathcal{X}, \mathcal{D}, \mathcal{R}, \mathcal{S}, \mathcal{T})$: each agent X_i does:

Time monitor: run continuously

1 **if** *deadline* t_i *reached* **then** commit to current best value: $X_i \leftarrow v_i^*$

2 **if** t_i *is hard commit* **then** mark X_i as *dead*; apply policy on dead agents

***UTIL* propagation protocol**: run continuously

3 **if** X_i *is hard-committed to* v_i* **then** $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)}[X_i = v_i*]$

4 **if** X_i *is soft-committed to* v_i* **then** $UTIL_{X_i}^{P(i)} = \left(JOIN_i^{P(i)} \oplus \sigma_i[v_i^*]\right) \perp_{X_i}$

5 **if** X_i *is not committed* **then** $UTIL_{X_i}^{P(i)} = JOIN_i^{P(i)} \perp X_i$

***VALUE* propagation protocol**

6 **if** X_i *is not committed* **then** $v_i^* \leftarrow argmax_{X_i}\left(JOIN_i^{P(i)}[agent\_view]\right)$

7 **if** X_i *is soft-committed* **then** $v_i^* \leftarrow argmax_{X_i}\left(JOIN_i^{P(i)}[agent\_view] \oplus \sigma_i[v_i^*]\right)$

10.4 Real time guarantees in dynamically evolving environments

In general, constraint optimization problems are NP-hard to solve, so it is difficult to provide real time guarantees. However, low impact faults as defined in Definition 36 are a particular case of changes which are easier to deal with: in a first phase, the agent touched by the low impact fault can almost instantly recompute its new optimal value (see Section 9.2.2.2). The agent then informs its neighbors of its assignment change via VALUE messages. In the worst case, this first phase requires sending all `n-1` VALUE messages. However, the VALUE propagation is very fast, as the messages are of linear size, and the processing required from each node when receiving a VALUE message is simply retrieving its best value which corresponds to this new assignment.

In a second phase, the algorithm needs to prepare itself for the next low-impact fault, by recomputing and retransmitting the new UTIL messages, and by computing the new utility vectors as in (see Section 9.2.2.2). This second phase may take much longer than the first one, as it may require much more computation, and it also may involve sending larger messages over the network, which is an expensive operation. In the worst case, a full UTIL propagation of `n-1` messages could be required to prepare the system for the next fault. However, assuming that during this time there appear no additional faults, the solution which is implemented in the first phase is already the optimal one, and thus the system is in the stable state.

Part V

Self-Interest

Chapter 11

Distributed VCG Mechanisms for Systems with Self-Interested Users

In this chapter we consider systems with self-interested users, which try to maximize their own utility. We focus on the efficient social choice problem (SCP)*, where the goal is to assign values, subject to side constraints, to a set of variables to maximize the total utility across a population of agents, where each agent has private information about its utility function.*

We show how to model SCP as a DCOP. Whereas existing DCOP algorithms can be easily manipulated by an agent, we introduce M-DPOP*, the first DCOP algorithm that provides a* faithful distributed implementation *for efficient social choice. Faithfulness ensures that no agent can benefit by unilaterally deviating from any aspect of the protocol, and is achieved by carefully integrating the Vickrey-Clarke-Groves (VCG) mechanism with DPOP. Determining agent i's payment requires solving the social choice problem without agent i. Here, we present a method to* reuse computation *performed in solving the main problem in a way that is robust against manipulation by the excluded agent. Experimental results show that as much as 87% of the computation required for solving the marginal problems can be avoided by re-use, providing very good scalability in the number of agents.*

Distributed optimization problems can model environments where a set of agents must agree on a set of decisions subject to side constraints. We consider settings in which each agent has its own preferences on subsets of these decisions. The agents are self interested, and each one would like to obtain the decision that maximizes its own utility. However, the system as a whole agrees (or some social designer determines) that a solution should be selected to maximize the total utility across all agents. Thus, this is a problem of *efficient social choice*. As motivation, we have in mind massively distributed problems such as meeting scheduling, where the decisions are about when and where to hold each meeting, or allocating airport landing slots to airlines, where the decisions are which airline is allocated which slot, or scheduling contractors in construction projects.

One approach to solve such problems is with a central authority that computes the optimal solution. In combination with an *incentive mechanism* such as the Vickrey-Clarke-Groves (VCG) mechanism[103], one can also prevent manipulation by misreporting preferences. However, in many practical settings it is hard to bound the problem so that such a central authority is feasible. Consider meeting scheduling: while each agent only participates in a few meetings, it is in general not possible to find a set of meetings that has no further constraints with any other meetings and thus can be optimized separately. Similarly, contractors in a construction project simultaneously work on other projects, again creating an web of dependencies that is hard to optimize in a centralized fashion. Privacy concerns also favor decentralized solutions[90].

Algorithms for distributed constraint reasoning, such as ABT and AWC ([228]), AAS[197], DPOP[160] and ADOPT[141], can deal with large problems as long as the influence of each agent on the solution is limited to a bounded number of variables. However, the current techniques assume *cooperative agents*, and do not provide robustness against misreports of preferences or deviations from the algorithm by self-interested agents. This is a major limitation. In recent years, *faithful distributed implementation*[150] has been proposed as a framework within which to achieve a synthesis of the methods of (centralized) MD with distributed problem solving. Until now, distributed implementation has been applied to lowest-cost routing[72, 192], and policy-based routing[73], on the Internet, but not to efficient social choice, a problem with broad applicability.

This chapter brings the following contributions:

- We show how to model the problem of efficient social choice as a DCOP, and adapt the DPOP algorithm to exploit the local structure of the distributed model and achieve the same scalability as would be possible in solving the problem in a centralized fashion.

- We provide an algorithm whose first stage is to *faithfully* generate the DCOP representation from the underlying social choice problem. Once the DCOP representation is generated, the next stages of our *M-DPOP* algorithm are also faithful, and form an *ex post Nash* equilibrium of the induced non-cooperative game.

- In establishing that DCOP models of social choice problems can be solved faithfully, we observe that the *communication and information structure* in the problem are such that no agent can prevent the rest of the system, in aggregate, from correctly determining the marginal impact that allowing for the agent's (reported) preferences has on the total utility achieved by the other agents. This provides the generality of our techniques to other DCOP algorithms.

- Part of achieving faithfulness requires solving the DCOP with each agent's (reported) preferences ignored in turn, and doing so without this agent able to interfere with this computational process. We provide an algorithm with this robustness property, that is nevertheless able to reuse, where possible, intermediate results of computation from solving the main problem that all agents.

- In experimental analysis, on a meeting scheduling problem that is a common benchmark in the literature, we demonstrate that as much as 87% of the computation required for solving the marginal problems can be avoided through reuse. In absolute numbers, this amounts to saving the computation associated with 1.96 million valuations out of a total of 2.25 million.

The M-DPOP algorithm defines a *strategy* for each agent in the extensive-form game induced by the DCOP for efficient social choice. In particular, the M-DPOP algorithm defines the messages that an agent should send, and the computation that an agent should perform, in response to messages received from other agents. In proving that M-DPOP forms a game-theoretic equilibrium, we show that no agent can benefit by unilaterally deviating, whatever the utility functions of other agents and whatever the constraints. Although not as robust as a *dominant strategy equilibrium*, because this (*ex post*) equilibrium requires every other agent to follow the algorithm, Parkes and Shneidman[150] have earlier commented that this appears to be the necessary "cost of decentralization."

It is worthwhile to note that while agents make payments to the bank as required by the VCG mechanism, the total payment made by each agent to the bank is always non-negative and M-DPOP *never runs at a deficit*.

The reuse of computation, in solving the marginal problems with each agent removed in turn, is especially important in settings of *distributed* optimization because motivating scenarios are those for which the problem size is massive, perhaps spanning multiple organizations and encompassing thousands of decisions. For example, consider project scheduling, inter-firm logistics, intra-firm meeting scheduling, etc. With appropriate problem structure, DCOP algorithms in these problems can scale linearly in the size of the problem. For instance, DPOP is able to solve such problems through a single back-and-forth traversal over the problem graph. But *without re-use the additional cost of solving each marginal problem would make the computational cost quadratic rather than linear* in the number of agents, which could be untenable in such massive-scale applications.

The rest of this chapter is organized as follows: we start with a background section on mechanism design in general. In Section 11.2 we formally introduce the social choice problem, we show how to model it as a DCOP, and present some examples. In Section 11.3 we describe an adaptation of the DPOP algorithm to our DCOP model of social choice problems. Section 11.4 introduces our model of self-interested agents and defines the (centralized) VCG mechanism. Section 11.4.4 provides a simple method, *Simple M-DPOP* to make DPOP faithful and serves to illustrate the excellent fit between the information and communication structure of DCOPs and faithful VCG mechanisms. In Section 11.5 we describe our main algorithm, M-DPOP, in which computation is re-used in solving the marginal problems with each agent removed in turn. We present experimental results in Section 11.5.3, and summarize M-DPOP in Section 11.5.4. Additionally, we provide a discussion on adapting other DCOP algorithms to achieve faithfulness in Section 11.6

11.1 Background on Mechanism Design and Distributed Implementation

This work draws on two research areas: distributed algorithms for constraint satisfaction and optimization, and mechanism design for coordinated decision making in multi-agent systems with self-interested agents. We briefly overview the most relevant results in these areas.

There is a long tradition of using *centralized* incentive mechanisms within Distributed AI, going back at least to Ephrati and Rosenschein[64] who considered the use of the VCG mechanism to compute joint plans; see also Sandholm[187] and Parkes et al.[149] for more recent discussions. Also noteworthy is the work of Rosenschein and Zlotkin[181, 244] on *rules of encounter*, which provided non-VCG based approaches for task allocation in systems with two agents.

On the other hand, there are very few known methods for *distributed problem solving* in the presence of self-interested agents. For example, the TRACONET[186] and the CONTRACTNET[46] systems are negotiation-based, distributed task reallocation allocation mechanisms. Nevertheless, neither TRACONET or CONTRACTNET were studied in the presence of game-theoretic agents, but only for simple, myopically-rational agent behaviors. This lack of thorough analysis holds for more recent works[63, 152, 153] as well. Similarly, Wellman's work on *market-oriented programming*[218, 219] considers the role of virtual markets in the support of optimal resource allocation, but is developed for a model of "price-taking" agents (i.e. agents that treat current prices as though they are final), rather than game-theoretic agents.

Izmalkov et al.[102] adopt cryptographic primitives such as *ballot boxes* to show how to convert *any* centralized mechanisms into a DI on a *fully connected* communication graph. There interest is in demonstrating the theoretical possibility of "ideal mechanism design" without a trusted center. They focus on the issue of *trust*: can mechanism design be performed without a trusted center? Our work has a very different focus: we seek computational tractability, do not require fully connected communication graphs, and make no appeal to cryptographic primitives. On the other hand, we are content to retain desired behavior in *some* equilibrium (remaining consistent with the MD literature) while Izmalkov et al. avoid the introduction of any additional equilibria beyond those that exist in a centralized mechanism.

In a similar line of work, Yokoo, Suzuki and Hirayama[204, 230–234] resort to cryptographic mechanisms to address incentive issues.[232] shows how to implement a combinatorial auction mechanism in a distributed fashion, such that the VCG outcome is selected. Their approach has some drawbacks, however: it requires the computation of prices for each possible bundle, for all bidders, i.e. $n \times 2^m$ prices, where n is the number of bidders, and m is the number of items. This, coupled with the fact that the computation of each price involves heavy cryptographic computations, limit the practical applicability of their approach.

The first step in providing a more satisfactory synthesis of distributed algorithms with MD was provided by the agenda of *distributed algorithmic mechanism design* (DAMD), due to Feigenbaum and colleagues[72, 74]. They consider the problem of lowest-cost interdomain routing on the Internet, and provide an efficient algorithm that computes the VCG outcome. The agents– in this case *autonomous systems* running network domains –could therefore not benefit by misreporting information about their own transit costs. However, they do not consider the robustness of the *algorithm itself* to manipulation. This problem is later fixed in[150], where the concept of *Distributed implementation* is introduced, which specifies this additional requirement. Parkes and Shneidman[150] provide the *partition principle* for achieving faithfulness [1] in an *ex post* Nash equilibrium. They do not provide however a concrete instantiation of their mechanism for social choice problems.

Ours is the first work to achieve faithfulness for general DCOP algorithms, demonstrated here via application to efficient social choice.

11.2 Social Choice Problems

We assume that the social choice problem consists of a finite but possibly large number of decisions that all have to be made at the same time. Each decision is modeled as a variable that can take values in a discrete and finite domain. Each agent has private information about the variables on which it places *relations*. Each relation associated with an agent defines the utility of that agent for each possible assignment of values to the variables in the domain of the relation. There may also be hard constraints that restrict the space of feasible joint assignments to subsets of variables.

Definition 39 (Social Choice Problem - SCP) *An efficient social choice problem can be modeled as a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ *such that:*

- $\mathcal{X} = \{X_1, ..., X_m\}$ *is the set of* ***public decision variables*** *(e.g. when and where to hold meetings, to whom should resources be allocated, etc);*
- $\mathcal{D} = \{d_1, ..., d_m\}$ *is the set of finite* ***public domains*** *of the variables* $\mathcal{X}$ *(e.g. list of possible time slots or venues, list of agents eligible to receive a resource, etc);*
- $\mathcal{C} = \{c_1, ..., c_q\}$ *is a set of* ***public constraints*** *that specify the feasible combinations of values of the variables involved. A* **constraint** c_j *is a function* $c_j : d_{j_1} \times .. \times d_{j_k} \rightarrow \{-\infty, 0\}$ *that returns 0 for all allowed combinations of values of the involved variables, and* $-\infty$ *for disallowed ones. We denote by* $scope(c_j)$ *the set of variables associated with constraint* c_j*;*

[1] An algorithm is *faithful* if an agent cannot benefit by deviating from any of its required actions, including information-revelation, computation and message passing.

- $\mathcal{A} = \{A_1, ..., A_n\}$ *is a set of* ***self-interested agents*** *involved in the optimization problem;* $X(A_i) \subseteq \mathcal{X}$ *is a (***privately known***) set of variables in which agent* A_i *is "interested" and on which it has relations.*

- $\mathcal{R} = \{R_1, ..., R_n\}$ *is a set of* ***private relations****, where* R_i *is the set of relations specified by agent* A_i *and relation* $r_i^j \in R_i$ *is a function* $r_i^j : d_{j_1} \times .. \times d_{j_k} \rightarrow \mathbb{R}$ *specified by agent* A_i*, which denotes the utility* A_i *receives for all possible values on the involved variables* $\{j_1, \ldots, j_k\}$ *(negative values mean costs). We denote by* $scope(r_i^j)$ *the domain of variables that* r_i^j *is defined on.*

The private relations of each agent may, themselves, be induced by the solution to local optimization problems on additional, private decision variables and with additional, private constraints. These are kept local to an agent and are not part of the SCP definition.

The optimal solution to the SCP is a complete instantiation X^* of all variables in $\mathcal{X}$, s.t.

$$X^* \in \arg\max_{X \in \mathcal{D}} \sum_{i \in \{1,..,n\}} R_i(X) + \sum_{c_j \in \mathcal{C}} c_j(X), \tag{11.1}$$

where $R_i(X) = \sum_{r_i^j \in R_i} r_i^j(X)$ is agent A_i's *total utility* for assignment X. This is the natural problem of social choice: the goal is to find a solution that maximizes the total utility of all agents, while respecting hard constraints; notice that the second sum is $-\infty$ if X is infeasible and precludes this outcmoe. We assume throughout that there is a feasible solution.

In introducing the VCG mechanism in Section 11.4.1 and onwards, we will require the solution to the SCP with the influence of each agent's relations removed in turn. For this, let $SCP(\mathcal{A})$ denote the main problem in Eq. (11.1), and we define a *marginal problem* as follows:

Definition 40 ($SCP(-A_i)$**: the *marginal problem without agent* A_i*)** *We call "the marginal problem without agent* A_i*", and we denote by* $SCP(-A_i)$*, the problem* $\max_{X \in \mathcal{D}} \sum_{j \neq i} R_j(X) + \sum_{c_j \in \mathcal{C}} c_j(X)$*. Note that all decision variables remain. The only difference between* $SCP(\mathcal{A})$ *and* $SCP(-A_i)$ *is that the preferences of agent* A_i *are ignored in solving* $SCP(-A_i)$*.*

For variable X_j, we refer to the agents A_i for which $X_j \in X(A_i)$ as forming the **community** for X_j.

Assumptions We choose to emphasize the following assumptions:

- Each agent knows the variables in which it is interested, together with the domain of any such variable and the hard constraints that involve the variable.

- Each decision variable is supported by a *community mechanism* that allows all interested agents to report their interest and learn about each other. For example, such a mechanism can be implemented using a bulletin board.

- For each constraint $c_j \in \mathcal{C}$, every agent A_k in a community $X_l \in scope(c_j)$, i.e. with $X_l \in X(A_k)$, can read the membership lists of all other communities $X_m \in scope(c_j)$ for $X_m \neq X_l$. In other words, every agent involved in a hard constraint knows about all other agents involved in that hard constraint.

- Each agent can communicate directly with all agents in all communities in which it is a member, and with all other agents involved in the same shared hard constraints. No other communication between agents is required.

In Section 11.4 we will establish that the step of identifying the SCP, via the community mechanism, is itself *faithful* so that self-interested agents will choose to volunteer the communities of which they are a member (and only those communities.)

11.2.1 Modeling Social Choice as Constraint Optimization

We first introduce a centralized, constraint optimization problem (COP) model of the efficient social choice problem. This model is represented as a *centralized problem graph*. Given this, we then model this as a distributed constraint optimization problem (DCOP), along with an associated *distributed problem graph*. The distributed problem graph makes explicit the control structure of the distributed algorithm that is ultimately used by the multi-agent system to solve the problem. Both sections are illustrated by reference to a meeting scheduling problem, as described in Section 2.3.1, and[127]. We now introduce the idea of self-interest in this problem: although the organization as a whole desires to minimize the cost of the whole process, each department and employee is self interested in that it wishes to maximize its own utility. An artificial currency is created for this purpose and a weekly assignment is made to each employee. Employees express their preferences for meeting schedules in units of this currency.

11.2.1.1 A Centralized COP Model as a MultiGraph

Viewed as a centralized problem, the SCP can be defined as a constraint optimization problem on a *multigraph*, i.e. a graph in which several distinct edges can connect the same set of nodes. We denote this $COP(\mathcal{A})$, and provide an illustration in Figure 11.1(a) in the meeting scheduling domain. The decision variables are the nodes, and relations defined over subsets of the variables form edges of the multigraph; *hyper*edges that connect more than two vertices at once in the case of a relation involving more than two variables. There can be multiple edges that involve the same set of variables, with each

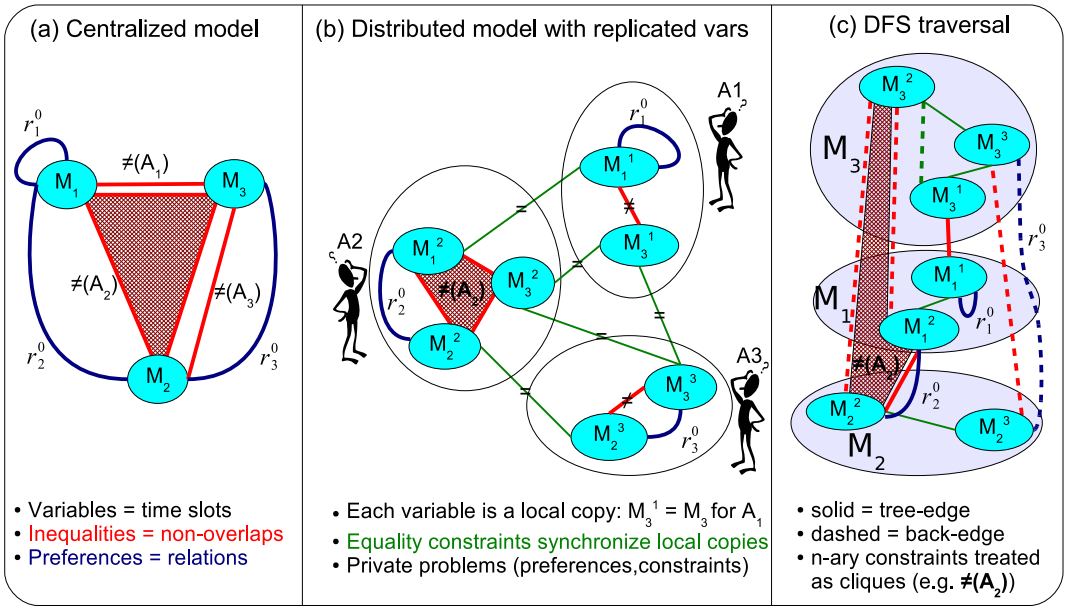


Figure 11.1: A meeting scheduling problem. **(a)** A centralized model: each vertex is a meeting variable, red edges correspond to hard constraints of non-overlap for meetings that share a participant [2], and blue edges correspond to relations and represent agent preferences. **(b)** A decentralized (DCOP) model with replicated variables: each agent has a local replica of variables of interest and green edges denote equality constraints that ensure agreement. The hard constraint for non-overlap between meetings M_1, M_2 and M_3 is now a local hyperedge to agent A_2. **(c)** A DFS arrangement of the decentralized problem graph: used by the DPOP algorithm to control the order of problem solving.

edge corresponding to the relations of a distinct agent on the same set of variables. The hard constraints are also be represented as edges on the graph.

Example 19 (Centralized Model for Meeting Scheduling) *The example in Figure 11.1(a) contains 3 agents and considers 3 meetings. The meetings* $\{M_1, M_2, M_3\}$ *correspond to the decision variables and the domain of each meeting is the available* time slots *for that meeting. Each vertex is associated with a meeting. Agent 1 must participate in meetings* M_1 *and* M_3*, agent 2 in every meeting, and agent 3 in meetings* M_2 *and* M_3*. These hard constraints are annotated as an edge for each of agents* A_1 *and* A_3 *and a hyperedge for agent* A_2*. Agent 1 expresses a relation on the of meeting* M_1*, agent 2 on the joint times assigned to meetings* M_1 *and* M_2 *and agent 3 on the joint times on* M_2 *and* M_3*. These relations are denoted with three edges on the graph, with the unary relation of agent 1 associated with a self-edge on vertex* M_1*.*

11.2.1.2 A Decentralized COP (DCOP) Model Using Replicated Variables

It is useful to define an alternate graphical representation of the SCP, with the centralized problem graph replaced with a *distributed* problem graph. This distributed problem graph has a direct correspondence

with the DPOP algorithm for solving DCOPs. We show in the following how to translate a SCP into a DCOP model.

Remark 10 *Both $SCP(\mathcal{A})$ (the problem with all agents included) and $SCP(-A_i)$ (the problem with agent A_i removed) can thus be translated into DCOP problems, which we denote by $DCOP(\mathcal{A})$ and $DCOP(-A_i)$, respectively.*

In our distributed model, each agent has a *local replica* of the variables in which it is interested.[3] For each public variable, $X_v \in X(A_i)$, in which agent A_i is interested, the agent has a *local replica*, denoted X_v^i. Agent A_i then models its local problem $COP(X(A_i), R_i)$, by specifying its relations $r_i^j \in R_i$ on the locally replicated variables.

The *neighborhood* of each local copy X_v^i of a variable is composed of three kinds of variables:

$$Neighbors(X_v^i) = Siblings(X_v^i) \cup Local\_neighbors(X_v^i) \cup Hard\_neighbors(X_v^i). \tag{11.2}$$

The siblings are local copies of X_v that belong to other agents $A_j \neq A_i$ also interested in X_v:

$$Siblings(X_v^i) = \{X_v^j \mid A_j \neq A_i \text{ and } X_v \in X(A_j)\} \tag{11.3}$$

All siblings of X_v^i are connected pairwise with an *equality constraint*. This ensures that all agents eventually have a consistent value for each variable. The second set of variables are the local neighbors of X_v^i from the local optimization problem of A_i. These are the local copies of the other variables that agent A_i is interested in, which are connected to X_v^i via relations in A_i's local problem:

$$Local\_neighbors(X_v^i) = \{X_u^i \mid X_u \in X(A_i), \text{ and } \exists r_i^j \in R_i \text{ s.t. } X_u^i \in scope(r_i)\} \tag{11.4}$$

We must also consider the set of *hard constraints* that contain in their scope the variable X_v and some other public variables: $Hard(X_v) = \{\forall c_s \in C | X_v \in scope(c_s)\}$. These constraints connect X_v with all the other variables X_u that appear in their scope, which may be of interest to some other agents as well. Consequently, X_v^i should be connected with all local copies X_t^j of the other variables X_t that appear in these hard constraints:

$$Hard\_neighbors(X_v^i) = \{X_t^j | \exists c_s \in Hard(X_v) \text{ s.t. } X_t \in scope(c_s), \text{ and } X_t \in X(A_j)\} \tag{11.5}$$

[3] An alternate model designates an "owner" agent for each decision variable. Each owner agent would then centralize and aggregate the preferences of other agents interested in its variable. Subsequently, the owner agents would use a distributed optimization algorithm to find the optimal solution. This model limits the reusability of computation from the main problem in solving the marginal problems in which each agent is removed in turn because when excluding the owner agent of a variable, one needs to assign ownership to another agent and restart the computational process in regards to this variable and other connected variables. This reuse of computation is important in making M-DPOP scalable. Our approach is disaggregated and facilitates greater reuse.

In general, each agent can also have *private variables*, and relations or constraints that involve private variables, and link them to the public decision variables. For example, consider a meeting scheduling application for employees of a company. Apart from the work-related meetings they schedule together, each one of the employees also has personal items on her agenda, like appointments to the doctor, etc. Decisions about the values for private variables and information about these local relations and constraints remain private. These provide no additional complications and will not be discussed further.

Example 20 (DCOP Model for Meeting Scheduling) *Refer to Figure 11.1 (b). In the example, each agent has as local variables the time slots corresponding to the meetings it participates in (e.g.* M_1^2 *represents* A_2*'s local replica of the variable representing meeting* M_1*). Local edges correspond to local all-different constraints between an agent's variables and ensure that it does not participate in several meetings at the same time. Equality constraints between local replicas of the same value ensure global agreement. Agents specify their relations via local edges on local replicas. For example, agent* A_1 *with its relation on the time of meeting* M_1 *can now express a preference for a meeting later in the day with relation* r_1^0*, which can assign low utilities to morning time slots and high utilities to afternoon time slots. Similarly, if* A_2 *prefers holding meeting* M_2 after *meeting* M_1*, then it can use the local relation* r_2^0 *to assign high utilities to all satisfactory combinations of timeslots and low utility otherwise. For example,* $\langle M_1 = 9AM, M_2 = 11AM\rangle$ *gets utility 10, and* $\langle M_1 = 9AM, M_2 = 8AM\rangle$ *gets utility 2.*

We can understand the potential for manipulation by self-interested agents through this example:

Example 21 (Manipulation Example) *Notice that although the globally optimal solution may require holding meeting* M_2 before *meeting* M_1*, this is less preferable to* A_2*, providing utility 2 instead of 10. Therefore, in the absence of an incentive mechanism,* A_2 *could benefit from a simple manipulation: declare utility* $+\infty$ *for* $\langle M_1 = 9AM, M_2 = 11AM\rangle$*, thus changing the final assignment to a suboptimal one that is nevertheless better for itself.*

11.3 Cooperative Case: Efficient Social Choice via DPOP

In this section, we instantiate DPOP for efficient social choice problems. Specifically, we first show in Section 11.3.1 how the optimization problem is constructed from the agents' interests in variables and their preferences. Subsequently, we show the changes we make to DPOP to adapt it to the SCP domain. The most prominent such adaptation exploits the fact that several variables represent local replicas of the same variable, and can be treated as such both during the UTIL and the VALUE phases. This adaptation improves efficiency significantly, and allows *complexity claims to be stated in terms of the induced width of the centralized COP problem graph rather than the distributed COP problem graph (see Section 11.3.5).*

11.3.1 Building the DCOP

To initialize the algorithm, each agent first forms the communities around its variables of interest, $X(A_i)$, and defines a local optimization problem $COP_i(X(A_i), R_i)$ with a replicated variable X_v^i for each $X_v \in X(A_i)$. Shorthand $X_v^i \in COP_i$ denotes that agent A_i has a local replica of variable X_v. Each agent owns multiple variables and we can conceptualize each variable as having an associated "virtual agent" operated by the owning agent. Each such virtual agent is responsible for the associated variable.

All agents subscribe to the communities in which they are interested, and learn which other agents belong to these communities. Neighboring relations are established for each local variable according to Eq. 11.2, as follows: all agents in a community X_v connect their corresponding local copies of X_v with equality constraints. By doing so, the local problems $COP_i(X(A_i), R_i)$ are connected with each other according to the interests of the owning agents. Local relations in each $COP_i(X(A_i), R_i)$ connect the corresponding local variables. Hard constraints connect local copies of the variables they involve. Thus, the overall problem graph $DCOP(\mathcal{A})$ is formed.

For example, consider again Figure 11.1(b). The decision variables are the start times of the three meetings. Each agent models its local optimization problem by creating local copies of the variables in which it is interested and expressing preferences with local relations. Formally, the initialization process is described in Algorithm 23.

Algorithm 23 *DPOP init: community formation and building* $DCOP(\mathcal{A})$.

DPOP_init($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}$):

1 Each agent A_i models its interests as $COP_i(X(A_i), R_i)$: a set of relations R_i imposed on a set $X(A_i)$ of variables X_v^i that each replicate a public variable $X_v \in X(A_i)$
2 Each agent A_i subscribes to the communities of $X_v \in X(A_i)$
3 Each agent A_i connects its local copies $X_v^i \in X(A_i)$ with the corresponding local copies of other agents via equality constraints

11.3.2 Constructing the DFS traversal

The method for DFS traversal is described in Algorithm 24. The algorithm starts by choosing one of the variables, X_0, as the root. This can be done randomly, for example using a distributed algorithm for random number generation, with a leader election algorithm (e.g.[147]), or by simply picking the variable with the lowest ID. The agents involved in the community for X_0 then randomly choose one of them, A_r as the *leader*. The local copy X_0^r of variable X_0 becomes the root of the DFS.

Once a root has been chosen, the agents participate in a *distributed depth-first traversal of the problem graph*. For convenience, we describe the DFS process as a token-passing algorithm in which all members within a community can observe the release or pick up of the token by the other agents. The

Algorithm 24 *DPOP Phase One: DFS construction.*

Inputs: each A_i knows its COP_i, and $Neighbors(X_v^i), \forall X_v^i \in COP_i$
Outputs: each A_i knows $P(X_v^i)$, $PP(X_v^i)$, $C(X_v^i)$, $PC(X_v^i), \forall X_v^i \in COP_i$.

Procedure Initialization
1 The agents choose one of the variables, X_0, as the root.
2 Agents in X_0's community elect a "leader", A_r.
3 A_r initiates the token passing from X_0^r to construct the DFS

Procedure Token Passing (performed by each "virtual agent" $X_v^i \in COP_i$)
4 **if** X_v^i *is root* **then** $P(X_v^i) = null$; create empty token $DFS := \emptyset$
5 **else** DFS:=**Handle_incoming_tokens()**
6 Let $DFS := DFS \cup \{X_v^i\}$
7 Sort $Neighbors(X_v^i)$ by $Siblings(X_v^i)$, then $Local\_neighbors(X_v^i)$, then $Hard\_neighbors(X_v^i)$. Set $C(X_v^i) := null$.
8 **forall** $X_l \in Neighbors(X_v^i)$ *s.t.* X_l *not visited yet* **do**
9 $\quad C(X_v^i) := C(X_v^i) \cup X_l$. Send DFS to X_l wait for DFS token to return.
10 Send DFS token back to $P(X_v^i)$.

Procedure Handle_incoming_tokens() //run by each "virtual agent" $\boldsymbol{X_v^i \in COP_i}$
11 Wait for any incoming DFS message; let X_l be the sender
12 Mark X_l as *visited*.
13 **if** *this is the first* DFS *message (i.e.* X_l *is my parent)* **then**
14 $\quad P(X_v^i) := X_l$; $PP(X_v^i) := \{X_k \neq X_l | X_k \in Neighbors(X_v^i) \cap DFS\}$; $PP(X_v^i) := \emptyset$
else
15 $\quad$ **if** $X_l \notin C(X_v^i)$ *(i.e. this is a DFS coming from a pseudochild)* **then**
$\quad\quad PC(X_v^i) := PC(X_v^i) \cup X_l$

neighbors of each node are sorted (in line 7) to prioritize for copies of variables held by other agents, and then other local variables, and finally other variables linked through hard constraints. Making the assumption that virtual agents act on behalf of each variable in the problem, the functioning of the token passing mechanism is similar to that described in Section 3.4.1.1.

Example 22 *Consider the meeting scheduling example in Figure 11.1. Assume that* M_3 *was chosen as the start community and* A_2 *was chosen within the community as the leader.* A_2 *creates an empty token* $DFS = \emptyset$ *and adds* M_3^2*'s ID to the token (*$DFS = \{M_3^2\}$*). As in Eq. 11.2,* $Neighbors(M_3^2) = \{M_3^3, M_3^1, M_1^2, M_2^2\}$. A_2 *sends the token* $DFS = \{M_3^2\}$ *to the first unvisited neighbor from this list, i.e.* M_3^3*, which belongs to* A_3. A_3 *receives the token and adds its copy of* M_3 *(now* $DFS = \{M_3^2, M_3^3\}$*).* A_3 *then sends the token to* M_3^3*'s first unvisited neighbor,* M_3^1 *(which belongs to* A_1*).*

Agent A_1 *receives the token and adds its own copy of* M_3 *to it (now* $DFS = \{M_3^2, M_3^3, M_3^1\}$*).* M_3^1*'s neighbor list is* $Neighbors(M_3^1) = \{M_3^2, M_3^3, M_1^1\}$*. Since the token that* A_1 *has received already*

contains M_3^2 and M_3^3, this means that they were already visited. Thus, the next variable to visit is M_1^1, which happens to be a variable that also belongs to A_1. The token is "passed" to M_1^1 internally (no message exchange required), and M_1^1 is added to the token (now $DFS = \{M_3^2, M_3^3, M_3^1, M_1^1\}$).

The process continues, exploring sibling variables from each community in turn, and then passing on to another community, and so on. Eventually all replicas of a variable are arranged in a chain and have equality constraints (back-edges) with all the predecessors that are replicas of the same variable. When a dead end is reached, the last agent backtracks by sending the token back to its parent. In our example, this happens when A_3 receives the token from A_2 in the M_2 community. Then, A_3 sends back the token to A_2 and so on. Eventually, the token returns on the same path all the way to the root and the process completes.

11.3.3 Handling the Public Hard Constraints.

Social choice problems, as defined in Definition 39 can contain side constraints, in the form of publicly known hard constraints, which represent domain knowledge such as "a resource can be allocated only once", "this hotel can accomodate 100 people", "no person can be in more than one meeting at the same time." etc. These constraints are not owned by any agent, but are available to all agents interested in any variable involved in the domain of any such constraint. Handling these constraints is essentially unchanged from handling the non-binary constraints in standard DPOP, as described in Section 3.4.1.1 for the DFS construction phase, and in Section 7 for the UTIL phase. Specifically:

DFS construction: neighboring relationships as defined in Eq. 11.2 require for each local variable that other local copies that share a hard constraint are considered as neighbors. This ensures that during the DFS construction phase, hard constraints are handled as any non-binary constraint, i.e. as a clique of the involved variables. Furthermore, in Algorithm 24, because of the prioritization in line 7, the DFS traversal is mostly made according to the structure defined by the relations of the agents and most hard constraints will appear as backedges in the DFS arrangement of the problem graph.

UTIL propagation: similarly to a non-binary constraint in DPOP, hard constraints are introduced in the UTIL propagation phase by the lowest agent in the community of the variable from the scope of the hard constraint, i.e. the agent with the variable that is lowest in the DFS ordering. For example, if there was a constraint between M_2 and M_3 in Figure 11.1 to specify that M_2 should occur after M_3 then this becomes a backedge between the 2 communities and would be assigned to A_3 for handling.

11.3.4 Handling replica variables

Our distributed model of SCP replicates each decision variable for every interested agent and connects all these copies with equality constraints. This in turn may increase the induced width k of the DCOP model with replicated variables when compared to the induced width w of the centralized model. This is best avoided, because DPOP's message size and computational complexity is exponential in the induced width. Specifically, with no further adaptation, the *UTIL* messages in DPOP on the distributed problem graph would be conditioned on as many variables as there are local copies of an original variable. However, all the local copies represent the same varible and must be assigned the same value; thus, sending many combinations where different local copies of the same variable take different values is wasteful. Therefore, we handle multiple replicas of the same variable in *UTIL* propagation as though they are the single, original variable, and condition UTIL messages on just this one variable. This is realized by updating the JOIN operator as follows:

Definition 41 (Updated JOIN operator for SCP) *Defined in two steps:*

Step 1: Consider all UTIL messages received as in input. For each one, consider each variable X_v^i on which the message is conditioned, and that is also a local copy of an original variable X_v. Rename X_v^i from the input UTIL message as X_v, i.e. the corresponding name from the original problem.

Step 2: Apply the normal JOIN operator for DPOP.

Applying the updated JOIN operator makes all local copies of the same variable become indistinguishable from each other, and merges them into a single dimension in the *UTIL* message and avoids this exponential blow-up.

Example 23 *Consider the meeting scheduling example in Figure 11.1. The centralized model in Figure 11.1(a) has a DFS arrangement that yields induced width 2 because it is a clique with 3 nodes. Nevertheless, the corresponding DCOP model in Figure 11.1(b) has induced width 3, as can be seen in the DFS arrangement from Figure 11.1(c), in which $Sep_{M_2^2} = \{M_3^2, M_3^3, M_1^2\}$. Applying DPOP to this DFS arrangement, M_2^2 would condition its UTIL message $UTIL_{M_2^2 \to M_1^2}$ on all variables in its separator: $\{M_3^2, M_3^3, M_1^2\}$. However, both M_3^2 and M_3^3 represent the same variable, M_3. Therefore, M_2^2 can apply the updated JOIN operator, which leverages the equality constraint between the two local replicas and collapse them into a single dimension (called M_3) in its message for M_1^2. The result it that the outgoing message only has 2 dimensions: $\{M_3, M_1^2\}$, and it takes much less space. This is possible because all 3 agents involved, i.e. A_1, A_2 and A_3 know that M_3^1, M_3^2 and M_3^3 represent the same variable.*

With this change, the VALUE propagation phase is modified so that only the top most local copy of any variable solve an optimization problem and compute the best value, announcing this result to all the other local copies which then assume the same value.

11.3.5 Complexity Analysis of DPOP Applied to Social Choice

The special handling of replica variables avoids the possible artificial increase in complexity and allows DPOP applied to SCP to scale with the induced width of the *centralized* problem graph, and independently of the number of agents involved and in the number of local replica variables.

Specifically, consider a DFS arrangement for the centralized model of the SCP that is equivalent to the DFS arrangement for the DCOP model, where "equivalent" means that the original variables from SCP are visited in the same order in which their corresponding communities are visited during the distributed DFS construction. (Recall that the distributed DFS traversal described in Section 11.3.2 visits all local copies from a community from DCOP before moving on to the next community). Let w denote the induced width of this DFS arrangement of the centralized SCP. Similarly, let k denote the induced width of the DFS arrangement of the distributed model. Let $D = \max_m |d_m|$ denote the maximal domain of any variable. Then, we have the following:

Theorem 8 (DPOP Complexity for SCP) *The number of messages passed in DPOP in solving a SCP is $2m$, $(n-1)$ and $(n-1)$ for phases one, two and three respectively, where n and m are the number of nodes and edges in the DCOP model with replicated variables. The maximal number of utility values computed by any node in DPOP is $O(D^{w+1})$, and the largest UTIL message has $O(D^{w+1})$ entries, where w is the induced width of the **centralized** problem graph.*

PROOF. The first part of the claim (number of messages) follows trivially from Proposition 1. For the second part (message size and computation): given a DFS arrangement of a DCOP, applying Proposition 1 trivially gives that in the basic DPOP algorithm, the maximal amount of computation on any node is $O(D^{k+1})$, and the largest *UTIL* message has $O(D^k)$ entries, where k is the induced width of the DCOP problem graph. To improve this analysis we need to consider the special handling of the replica variables.

Consider the *UTIL* messages which travel up along the DFS tree, and whose sets of dimensions contain the separators of the sending nodes. Recall that the updated JOIN collapses all local replicas into the original variables. The union of the dimensions of the *UTIL* messages to join in the DPOP on the DCOP model becomes identical to the set of dimensions of the nodes in the DPOP on the centralized model. Thus, each node in the DCOP model performs the same amount of computation as its counterpart on the centralized model. It follows that the *computation* required in DPOP scales as $O(D^{w+1})$ rather than $O(D^{k+1})$ by this special handling.

There remains an additional difference between DPOP on the DFS arrangement for the centralized SCP versus DPOP on the DFS arrangement for the DCOP. A variable X_v that is replicated across multiple agents can only be projected out from the *UTIL* propagation through local optimization by the top-most agent handling a local replica of X_v. This is the first node at which all relevant information is in place to support this optimization step. In particular, whenever a node with the maximal separator

set is not also associated with the top-most replica of its variable then it must retain dependence on the value assigned to its variable in the *UTIL* message that it sends to its parent. This increases the worst case *message size* of DPOP to $O(D^{w+1})$, as opposed to $O(D^w)$ for the normal DPOP. Computation remains $O(D^{w+1})$ because the utility has to be determined for each value of X_v anyway, and before projecting X_v out. □

To see the effect described in the proof, in which a local variable cannot be immediately removed during *UTIL* propagation, consider again the problem from Figure 11.1. Suppose now that agent A_3 is also involved in meeting M_1. This introduces an additional back-edge $M_2^3 - M_1^3$ in the DFS arrangement for the decentralized model shown in Figure 11.1(c).

The DFS arrangement of the COP model that corresponds to the decentralized model is simply a traversal of the COP in the order in which the communities are visited during the distributed DFS construction. This corresponds to a chain: $M_3 - M_1 - M_2$. The introduction of the additional back-edge $M_2^3 - M_1^3$ in the distributed DFS arrangement does not change the DFS of the COP model, and its width remains $w = 2$. However, as M_2^3 is not the top most copy of M_2, agent A_3 cannot project M_2 out of its outgoing *UTIL* message. The result is that it sends a *UTIL* message with $w + 1 = 3$ dimensions, as opposed to just $w = 2$.

11.4 Handling Self-interest: A Faithful Algorithm for Social Choice

Having adapted DPOP to remain efficient for SCPs, we now turn to the issue of self-interest. Without further modification, an agent can manipulate DPOP by misreporting its private relations and deviating from the algorithm in various ways. In the setting of meeting scheduling, for example, an agent might benefit by *misrepresenting its local preferences* ("I have massively more utility for the meeting occurring at 2pm than at 9am"), *incorrectly propagating utility information of other (competing) agents* ("The other person on my team has very high utility for the meeting at 2pm"), or by *incorrectly propagating value decisions* ("It has already been decided that some other meeting involving the other person on my team will be at 9am so this meeting must be at 2pm.")

By introducing carefully crafted payments, by leveraging the information and communication structure inherent to DCOPs for social choice, and by careful partitioning of computation so that each agent is only asked to reveal information, perform optimization, and send messages that are in its own interest, we are able to achieve faithfulness. This will mean that each agent will *choose*, even when self-interested, to follow the modified algorithm.

We first define the VCG mechanism for social choice and illustrate its ability to prevent manipulation in centralized problem solving in a simple example. With this in place, we next review the definitions of *faithful distributed implementation* and the results of a useful principle, the *partition principle*. In closing this section, we then describe the *simple M-DPOP* algorithm and prove its faithfulness.

11.4.1 The VCG Mechanism Applied to Social Choice Problems

Mechanism design (MD) addresses the problem of optimizing some criteria, frequently social welfare, in the presence of self-interested agents that each have private information relevant to the problem at hand. In the standard story, agents report private information to a "center," that solves an optimization problem and enforces the outcome.

In our setting of efficient social choice, we will assume the existence of a *currency* so that agents can make payments, and make the standard assumption of *quasilinear* utility functions, so that agent A_i's *net utility* is,

$$u_i(X, p) = R_i(X) - p, \tag{11.6}$$

for an assignment $X \in \mathcal{D}$ to variables $\mathcal{X}$ and payment $p \in \mathbb{R}$ to the center, i.e., its net utility is that due to the decision, $R_i(X) = \sum_{r_i^j \in R_i} r_i^j(X)$, minus the amount of its payment.

One of the most celebrated results of MD is provided by the Vickrey-Clarke-Groves (VCG) mechanism. The VCG mechanism generalizes Vickrey's second price auction to the problem of efficient social choice:

Definition 42 (VCG mechanism for Efficient Social Choice) *Given knowledge of public constraints $\mathcal{C}$, and public decision variables $\mathcal{X}$, the mechanism works as follows:*

- *Each agent, A_i, makes a report $\hat{R}_i$ about its private relations.*
- *The center's decision, X^*, is that which solves $SCP(\mathcal{A})$* given the reports $\hat{R} = (\hat{R}_1, \ldots, \hat{R}_n)$.
- *Each agent A_i, makes payment*

$$Tax(A_i) = \sum_{j \neq i} \left(\hat{R}_j(X^*_{-i}) - \hat{R}_j(X^*) \right), \tag{11.7}$$

  to the center, where X^*_{-i}, for each A_i, is the solution to $SCP(-A_i)$ given reports $\hat{R}_{-i} = (\hat{R}_1, \ldots, \hat{R}_{i-1}, \hat{R}_{i+1}, \ldots, \hat{R}_n)$.

Each agent makes a payment that equals the *negative marginal externality that its presence imposes on the rest of the system*, in terms of influencing the solution to the SCP.

The VCG mechanism has a number of useful properties:

- **Strategyproofness:** Each agent's weakly dominant strategy, i.e. its utility-maximizing strategy whatever the strategies and whatever the private information of other agents, is to truthfully report its private relations to the center.

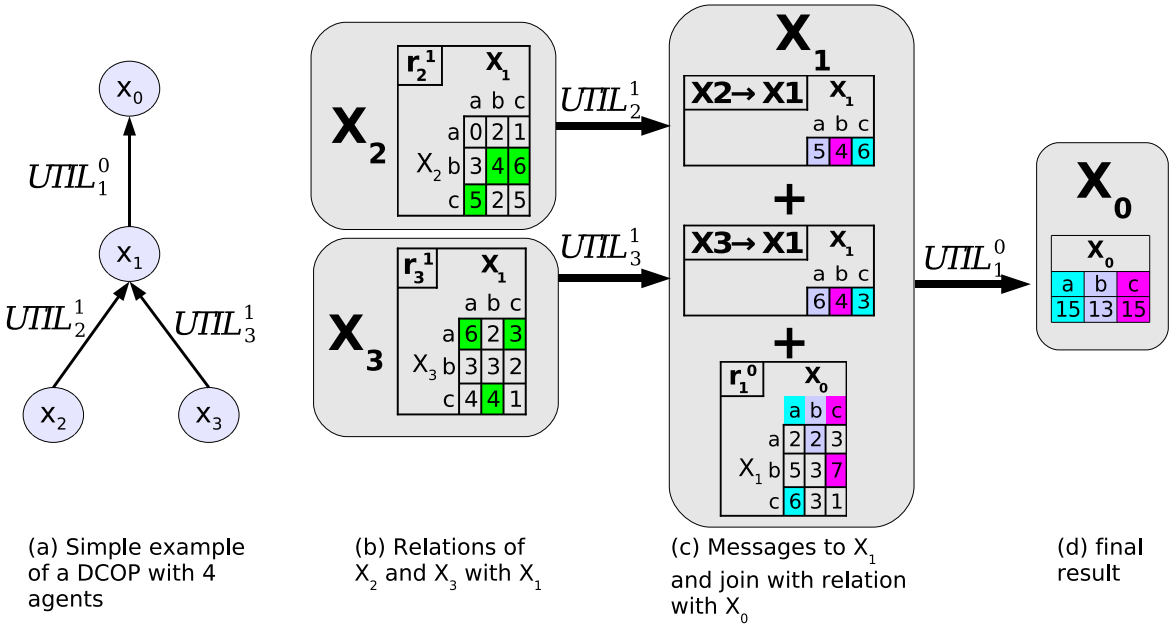


Figure 11.2: Numerical example of UTIL propagation. (a) A simple DCOP problem in which there are three relations r_3^1, r_2^1 and r_1^0 between (X_3, X_1), (X_2, X_1) and (X_1, X_0) respectively. (b) Projections of X_2 and X_3 out of their relations with X_1. The results are sent to X_1 as $UTIL_2^1$, and $UTIL_3^1$ respectively. (c) X_1 joins $UTIL_2^1$ and $UTIL_3^1$ with its own relation with X_0. (d) X_1 projects itself out of the join and sends the result to X_0.

- **Efficiency:** In equilibrium, the mechanism makes a decision that maximizes the total utility to agents over all feasible solutions to the SCP.
- **Participation:** In equilibrium, each agent's net utility, $R_i(X^*) - Tax(A_i) = (R_i(X^*) + \sum_{j \neq i} \hat{R}_j(X^*)) - \sum_{j \neq i} \hat{R}_j(X^*_{-i})$, is non-negative, by the principle of optimality, and therefore agents will choose to participate.
- **No-Deficit:** The payment made by each agent is non-negative in the SCP, because $\sum_{j \neq i} \hat{R}_j(X^*_{-i}) \geq \sum_{j \neq i} \hat{R}_j(X^*)$, by the principle of optimality, and therefore the entire mechanism runs at a budget surplus.

To understand why the VCG mechanism is strategyproof, notice that the first term in $Tax(A_i)$ is independent of A_i's report. Now, the second term when taken together with the agent's own *true* utility from the decision, provides A_i with net utility $R_i(X^*) + \sum_{j \neq i} \hat{R}_j(X^*)$. This is the total utility for all agents, and to maximize this the agent should simply report its true relations, because the center will then explicitly solve this problem in picking X^*.

Example 24 (A numerical example of VCG computation) *Consider the simple DCOP example in Figure 11.2. We can make this into a SCP by associating agents A_1, A_2 and A_3 with relations r_1^0, r_2^1 and r_3^1 on variables $\{X_0, X_1\}$, $\{X_1, X_2\}$, and $\{X_1, X_3\}$ respectively. Breaking ties as before, the*

solution to $SCP(\mathcal{A})$ is $< X_0 = a, X_1 = c, X_2 = b, X_3 = a >$ with utility $< 6, 6, 3 >$ to agents A_1, A_2 and A_3 respectively. Removing agent A_1, the solution would be $< X_0 =?, X_1 = a, X_2 = c, X_3 = a >$ with utility $< 5, 6 >$ to agents A_2 and A_3. The '?' indicates that agents A_2 and A_3 are indifferent to the value on X_0. Removing agent A_2, the solution would be $< X_0 = c, X_1 = b, X_2 =?, X_3 = c >$, with utility $< 7, 4 >$ to agents A_1 and A_3. Removing agent A_3, the solution would be $< X_0 = a, X_1 = c, X_2 = b, X_3 =? >$, with utility $< 6, 6 >$ to agents A_1 and A_2. The VCG mechanism would assign $< X_0 = a, X_1 = c, X_2 = b, X_3 = a >$, with payments $(5 + 6) - (6 + 3) = 2, (7 + 4) - (6 + 3) = 2, (6 + 6) - (6 + 6) = 0$ collected from agents A_1, A_2 and A_3 respectively. A_3 has no negative impact on agents A_1 and A_2 and does not incur a payment. The other agents make payments: the presence of A_1 helps A_2 but hurts A_3 by more, while the presence of A_2 hurts both A_1 and A_3. The only conflict in this problem is about the value assigned to variable X_1. Agents A_1, A_2 and A_3 each prefer that X_1 be assigned to b, c and a respectively. In the chosen solution, only agent A_2 gets its best outcome. Considering the case of A_3, it can force either a or b to be selected by reporting a suitably high utility for this choice, but for $X_1 = a$ it must pay 4 while for $X_1 = b$ it must pay 1, and in either case it weakly prefers the current outcome in which it makes zero payment.

In fact, there is a real sense in which we are *only* able to address self-interest in DCOPs by maximizing something like the total utility of participants. (More generally, it is straightforward to extend our techniques to maximize a *linear weighted sum* of the utility of each agent for the solution, where these weights are fixed and known, for instance by a social planner[103].) Roberts[179] proves that the Groves mechanisms are the *only*, non-trivial strategyproof mechanisms in the domain of social choice unless one makes additional assumptions about the structure of the domain; e.g., everyone prefers earlier meetings, or more of a resource is always weakly preferred to less.[4]

11.4.2 Faithful Distributed Implementation

Our goal here is to find a way to distribute the computation required to solve the SCP, and to determine the VCG payments, onto the agents while retaining an analog to strategyproofness. This is the problem of distributed implementation (DI), which seeks to distribute the computation performed by the center in the traditional model of MD to the agents. This is challenging because it opens up additional opportunities for manipulation beyond those in the centralized VCG.

Additional Assumptions we introduce the following additional assumptions over-and-above those made so far in Section 11.2:

- Agents are *rational but helpful*, meaning that although self-interested, they will follow a protocol

[4]Together with another technical assumption, Robert's theorem has been extended by Lavi, Mu'alem and Nisan[121] to domains that allow this kind of structure, for instance to combinatorial auctions.

whenever there is no deviation that will make them *strictly* better off (given the behavior of other agents).

- Each agent is prevented from posing as several independent agents by an external technique for providing strong (but perhaps pseudonymous) identities.
- *Catastrophic failure* will occur if all agents in the community of a variable do not eventually choose the same value for the variable.
- A trusted bank, connected with a *trusted communication channel* to each agent and with the authority to collect payments from each agent.

By a trusted communication channel, we mean that each agent can send a message to the bank without interference by any other agent. These messages are only sent upon termination of M-DPOP, to inform the bank about other agents' payments. The bank is the only trusted entity that we need to assume. We continue to assume that the SCP has a feasible solution (and therefore that each marginal problem also has a feasible solution.) Catastrophic failure ensures that the decision determined by the protocol is actually executed. It prevents a "hold-out" problem, where an unhappy agent refuses to adopt the consensus decision.[5]

Given a distributed algorithm (such as *simple M-DPOP*, to be introduced shortly), we formalize this, for the same of analysis as a distributed implementation (DI), $d_M =< g, \Sigma, \breve{s} >$, which is defined in terms of three components[150, 192]:

- A *strategy space*, Σ, for each agent A_i. This restricts the space of messages that an agent can send in every possible state of the distributed algorithm. Given a DI, the way to think about this is that the other agents will only be programmed (in equilibrium) to be able to intepret a particular, well-defined set of messages that agent A_i could send.
- A *strategy*, $\sigma_i \in \Sigma$, exactly defines the message(s) that agent A_i will send in every possible state of the distributed algorithm. By defining the message(s) that are sent this encompasses all computation performed internal to an agent, all information-revelation decisions made by an agent about its private information, and all decisions made by an agent about how to propagate information received as messages from other agents.
- A *suggested protocol*, $\breve{s} = (\breve{s}_1, \ldots, \breve{s}_n)$, defines a strategy $\breve{s}_i(R_i) \in \Sigma$, for every agent A_i and all possible private relations R_i. That is, a suggested protocol $\breve{s}_i$ for A_i defines the messages that A_i will send in all possible states of the distributed algorithm, and for all possible private inputs of the agent.
- A two-part *outcome rule*, $g = (g_1, g_2)$, where $g_1 : \Sigma^n \rightarrow \mathcal{D}$ defines the assignment of values, $g_1(\sigma) \in \mathcal{D}$, to variables $\mathcal{X}$ given a *joint strategy*, $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma^n$, and $g_2 : \Sigma^n \rightarrow \mathbb{R}^n$ defines the payment $g_{2,i}(\sigma) \in \mathbb{R}$ made by each agent A_i given joint strategy $\sigma \in \Sigma^n$.

[5] An alternative solution would be to have agents report the final decision to a trusted party, responsible for enforcement.

To provide an additional interpretation, one can think about protocol $\breve{s}$ as corresponding to the algorithm, such as simple M-DPOP, that one wishes to show is faithful. Coupled with the distributed input to the problem, $R = (R_1, \ldots, R_n)$ and the known parts of the input such as hard constraints $\mathcal{C}$, then algorithm $\breve{s}$ defines particular messages that will be sent in every possible state of the algorithm. It is these messages that are defined by a strategy σ, which defines a particular execution trace of the algorithm given the input, and in turn the outcome $g(\sigma)$, where $g_1(\sigma)$ is the assignment of values determined on termination and $g_2(\sigma)$ is the vector of payments to collect from agents.[6]

The main question that we ask, given a distributed algorithm in the presence of self-interested agents, is whether the algorithm is an *ex post* Nash equilibrium.

Definition 43 (Ex post Nash equilibrium.) *A protocol $s = (s_1, \ldots, s_n)$, that defines a strategy $s_i(R_i) \in \Sigma$ for each agent A_i, for all possible private relations R_i, is an ex post Nash equilibrium (EPNE) in this context of social choice, if*

$$R_i(g_1(s_i(R_i), s_{-i}(R_{-i}))) - g_2(s_i(R_i), s_{-i}(R_{-i})) \geq R_i(g_1(\sigma_i', s_{-i}(R_{-i}))) - g_2(\sigma_i', s_{-i}(R_{-i})),$$
$$\forall \sigma_i' \in \Sigma_i, \forall R_i, \forall R_{-i} \tag{11.8}$$

This is defined so that no agent A_i can benefit by deviating from protocol, s_i, whatever the particular instance of DCOP (i.e. for all private relations $R = (R_1, \ldots, R_n)$), *so long as the other agents also choose to follow the protocol*. It is this latter clause that makes EPNE weaker than dominant-strategy equilibrium, in which s_i would be the best protocol for agent i *even if* the other agents followed an arbitrary protocol. Given this, we can define a *faithful* DI:

Definition 44 (Faithful Distributed Implementation) *Distributed implementation $d_M =< g, \Sigma, \breve{s} >$ is ex post faithful, if suggested protocol, $\breve{s}$, is an ex post Nash equilibrium.*

That is, when a suggested protocol, or algorithm, $\breve{s}$, is *ex post* faithful (or just faithful) then it is in the best interest of every agent A_i to follow all aspects of the algorithm – information revelation, computation and message-passing – whatever the private inputs of the other agents, as long as every other agent follows the algorithm.

11.4.3 The Partition Principle applied to Efficient Social Choice

One cannot achieve a faithful DI for efficient SCP by simply running DPOP, $n + 1$ times on the same problem graph, once for the main problem and then with each agent's effect nullified in turn by asking

[6] Note that the outcome rule must be well-defined for any unilateral deviation from $\breve{s}$, i.e. where aby one agent deviates and does not follow the suggested protocol. Here we assume that either the protocol still reaches a terminal state so that decisions and payments are defined, or that the protocol reaches some "bad" state with suitably negative utility to all participants, such as livelock or deadlock. We neglect this latter possibility for the rest of our analysis, but it can be easily treated by introducing special notation for this bad outcome.

it to simply propagate messages. Agent A_i would seek to do the following: (a) interfere with the computational process for $SCP(-A_i)$, to make the solution as close as possible to that to $SCP(\mathcal{A})$, so that its marginal impact appears small; and (b) otherwise decrease its payment, for example by increasing the apparent utility of other agents for the solution to $SCP(\mathcal{A})$, and in turn increase the value of the second term in its VCG payment (Eq. 11.7).

This opportunity for manipulation was recognized by Parkes and Shneidman[150], who proposed the *partition principle* as a method for achieving faithfulness in distributed VCG mechanisms, instantiated here in the context of efficient social choice problems:

Definition 45 (partition principle) *A distributed algorithm, corresponding to suggested protocol $\breve{s}$, satisfies the partition principle in application to efficient social choice, if:*

1. **(Correctness)** *An optimal solution is obtained for $SCP(\mathcal{A})$ and $SCP(-A_i)$ when every agent follows $\breve{s}$, and the bank receives messages that instruct it to collect the correct VCG payment from every agent.*
2. **(Robustness)** *Agent A_i cannot influence the solution to $SCP(-A_i)$, or the report(s) that the bank receives about the negative externality that A_i imposes on the rest of the system conditioned on solutions to $SCP(\mathcal{A})$ and $SCP(-A_i)$.*
3. **(Enforcement)** *The decision that corresponds to $SCP(\mathcal{A})$ is enforced, and the bank collects the payments as instructed.*

Proposition 15 *[150] A distributed algorithm that satisfies the partition principle is an ex post faithful distributed implementation for efficient social choice.*

By the partition principle, no agent A_i is able to prevent the other agents from correctly solving $SCP(-A_i)$, and neither can the agent prevent the other agents correctly reporting the negative externality that A_i imposes on the other agents by its presence. On the other hand, no restriction is placed on the agent's ability to influence the decision to $SCP(\mathcal{A})$. For example, it is permissible for every agent to use the standard DPOP algorithm to solve the main social choice problem.

For some intuition behind this result, note that the opportunity for manipulation by an agent A_i is now restricted to: (a) influencing the solution computed to $SCP(\mathcal{A})$; and (b) influencing the payments made by other agents. As long as the other agents follow the algorithm, it the ex post faithfulness then follows from the strategyproofness of the VCG mechanism because the additional opportunity here is to change (either increase or reduce) the amount of some *other* agent's payment.

Remark 11 (Ex-post Nash equilibrium vs. dominant strategy) *As has been suggested in previous work, the weakening from dominant-strategy equilibrium in the centralized VCG mechanism, to* ex post

Nash equilibrium in a distributed implementation, can be viewed as the "cost of decentralization". The incentive properties necessarily rely on the computation performed, and thus the strategy followed, by the other agents.[7]

11.4.4 Simple M-DPOP

Algorithm 25 describes simple-M-DPOP. In this variation the main problem, $SCP(\mathcal{A})$ is solved, followed by the social choice problem, $SCP(-A_i)$ with each agent removed in turn.[8] Once these $n+1$ problems are solved, every agent A_j knows the local part of the solution to X^* and X^*_{-i} for all $A_i \neq A_j$, that is the part of the solution that affects its own utility. This is critical, because it provides enough information to allow the system of agents without some agent A_i, for any A_i, to each send a message to the bank about a *component* of the payment that agent A_i should make.

Algorithm 25 *Simple-M-DPOP.*

1 Run DPOP for $DCOP(\mathcal{A})$ on $DFS(\mathcal{A})$; find X^*
2 **forall** $A_i \in \mathcal{A}$ **do**
3 Build $DFS(-A_i)$; run DPOP for $DCOP(-A_i)$ on $DFS(-A_i)$; find X^*_{-i}
4 All agents $A_j \neq A_i$ compute $Tax_j(A_i) = R_j(X^*_{-i}) - R_j(X^*)$ and report it to the bank.
5 Bank deducts $\sum_{j \neq i} Tax_j(A_i)$ from A_i's account
6 Each A_i assigns values in X^* as the solution to its local COP_i

The computation of payments is disaggregated across the agents. The tax payment collected from agent A_i as a result of the message sent to the bank by agent A_j, is defined (in the truthful equilibrium) as:

$$Tax_j(A_i) = R_j(X^*_{-i}) - R_j(X^*), \tag{11.9}$$

which is defined so that $Tax(A_i) = \sum_{j \neq i} Tax_j(A_i)$. The value, $Tax_j(A_i)$, represents the payment made by agent A_i in the VCG mechanism as a result of its negative effect on the utility of agent A_j.

The important observation, in being able to satisfy the partition principle, is that these component of A_i's payment satisfies a **locality property**, so that *each agent A_j can compute this component of A_i's payment with just its private information about its relations and its local information about the part of solutions X^* and X^*_{-i} that affects its own utility*, all of which is available upon termination of DPOP in the main problem and in the problem without A_i. Correctly determining this payment,

[7] An exception is provided by Izmalkov et al.[102], who are able to avoid this through the use of cryptographic primitives, in their case best thought of as physical devices such as ballet boxes.

[8] Simple M-DPOP is presented for a setting in which the main problem and the subproblems are connected but extends immediately to disconnected problems. Indeed, it may be that the main problem is connected but one or more subproblems are disconnected. To see that there are no additional incentive concerns notice that it is sufficient to recognize that the correctness and robustness properties of the partition principle would be retained in this case.

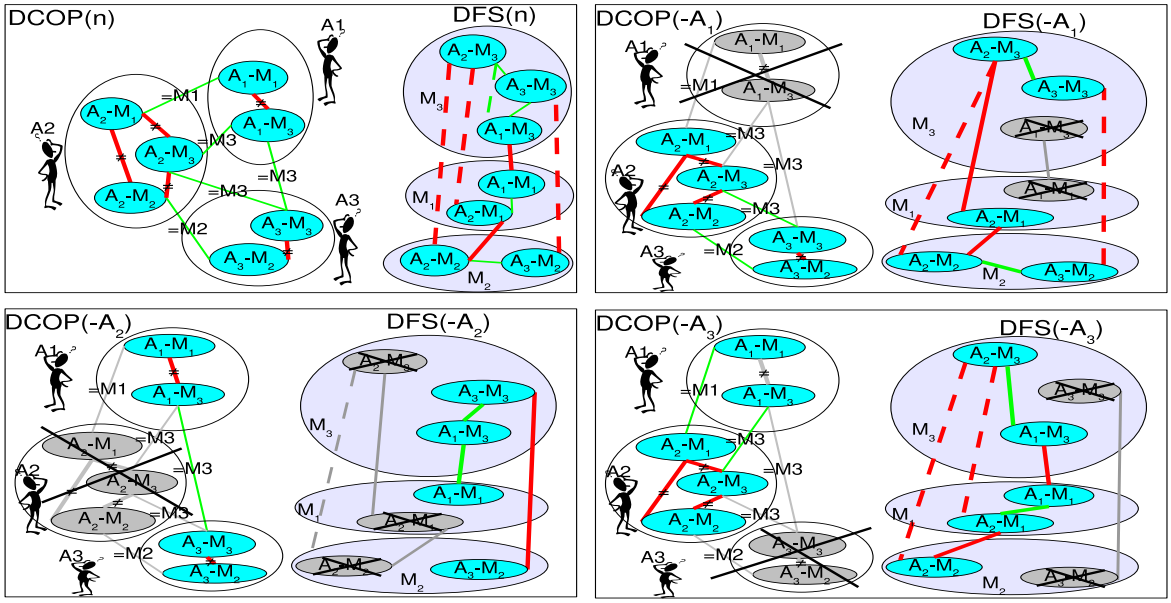


Figure 11.3: Simple M-DPOP: Each agent A_i is excluded in turn from the optimization $DCOP(-A_i)$. This is illustrated on the meeting scheduling example.

conditioned on X^* and X^*_{-i}, does not rely on any aspect of any other agent's algorithm, including that of A_i.[9]

Figure (11.3) provides an illustration of simple M-DPOP on the earlier meeting scheduling example, and shows how the marginal problems (and the DFS arrangements for each such problem) are related to the main problem.

Theorem 9 *The simple-M-DPOP algorithm is a faithful distributed implementation of efficient social choice and terminates with the outcome of the VCG mechanism.*

PROOF. To prove this we establish that simple-M-DPOP satisfies the partition principle. First, DPOP computes optimal solutions to $SCP(\mathcal{A})$ and $SCP(-A_i)$ for all $A_i \in \mathcal{A}$ when every agent follows the protocol. This is immediate because of the correctness of the DCOP model of SCP and the correctness of DPOP. The correct VCG payments are collected when every agent follows the algorithm by the correctness of the disaggregation of VCG payments in Eq. 11.9. Second, agent A_i cannot influence the solution to $SCP(-A_i)$ because it is not involved in that computation in any way. The DFS arrangement is constructed, and the problem solved, by the other agents, who completely ignore A_i and any messages that agent A_i might send. (Any hard constraints that A_i may have handled in $SCP(\mathcal{A})$ are reassigned automatically to some other agent in $SCP(-A_i)$ as a consequence of the fact that the

[9] A similar disaggregation was identified by Feigenbaum et al.[72] for lowest-cost interdomain routing on the Internet. Shneidman and Parkes[192] subsequently modified the protocol so that agents other than A_i had enough information to report the payments to be made by agent A_i.

DFS arrangement is reconstructed. DPOP still solves $SCP(-A_i)$ correctly in the case that the problem graph corresponding to $SCP(-A_i)$ becomes disconnected (in this case the DFS arrangement is a forest). The robustness of the value of the reports from agents $\neq A_i$ about the negative externality imposed by A_i, conditioned on solutions to $SCP(\mathcal{A})$ and $SCP(-A_i)$, follows from the locality property of payment terms $Tax_j(A_i)$ for all $A_j \neq A_i$. For enforcement, the bank is trusted and empowered to collect payments, and all agents will finally set local copies of variables as in X^* to prevent catastrophic failure. Agent A_i will not deviate as long as other agents do not deviate. Moreover, if agent A_i is the only agent that is interested in a variable then its value is already optimal for agent A_i anyway. □

The partition principle, and faithfulness, has sweeping implications. Not only will each agent follow the subtantive aspects of simple-M-DPOP, *but each agent will also chose to faithfully participate in the community discovery phase, in any algorithm for choosing a root community, and in selecting a leader agent in Phase one of DPOP.*[10]

Remark 12 (Antisocial behavior) *Note that reporting exaggerated taxes hurts other agents but does not increase one's own utility so this is excluded by our assumption that the agents are self-interested but helpful (see Section 11.4.2).*

11.5 M-DPOP: Reusing Computation While Retaining Faithfulness

In this section, we introduce the M-DPOP algorithm. In *simple-M-DPOP*, the computation to solve the main problem is completely isolated from the computation to solve each of the marginal problems. In comparison, in *M-DPOP* we re-use computation already performed in solving the main problem in solving the marginal problems. This enables the algorithm to scale well in practice to problems where each agent's influence is limited to a small part of the entire problem because little additional computation is required beyond that of DPOP.

The challenge that we face, in facilitating this re-use of computation, is to retain the incentive properties that are provided by the partition principle. *A possible new manipulation is for agent A_i to deviate in the computation in $DCOP(\mathcal{A})$, with the intended effect to change the solution to $DCOP(-A_i)$ via the indirect impact of the computation performed in $DCOP(\mathcal{A})$ when it is reused in solving $DCOP(-A_i)$. To prevent this, we have to determine which UTIL messages in $DCOP(\mathcal{A})$ could not have been influenced by agent A_i.*

Example 25 (Reusing computation safely based on problem structure) *Refer to Figure 11.4. Here*

[10] One can also observe that is not useful for an agent to misreport the local utility of *another agent* A_j while sending $UTIL$ messages around the system. On one hand, such a deviation could of course change the selection of X^* or X^*_{-k} for some $k \neq \{i, j\}$ and thus the payments by other agents or the solution ultimately selected. But, by deviating in this way the agent cannot change the utility information that is finally used in determining its own payments. This is because it is agent A_j *itself* that computes the marginal effect of agent A_i on its local solution, and component $Tax_j(A_i)$ of agent A_i's payment. Thus, we are able to protect against this manipulation through leveraging the disaggregated definition of VCG payments.

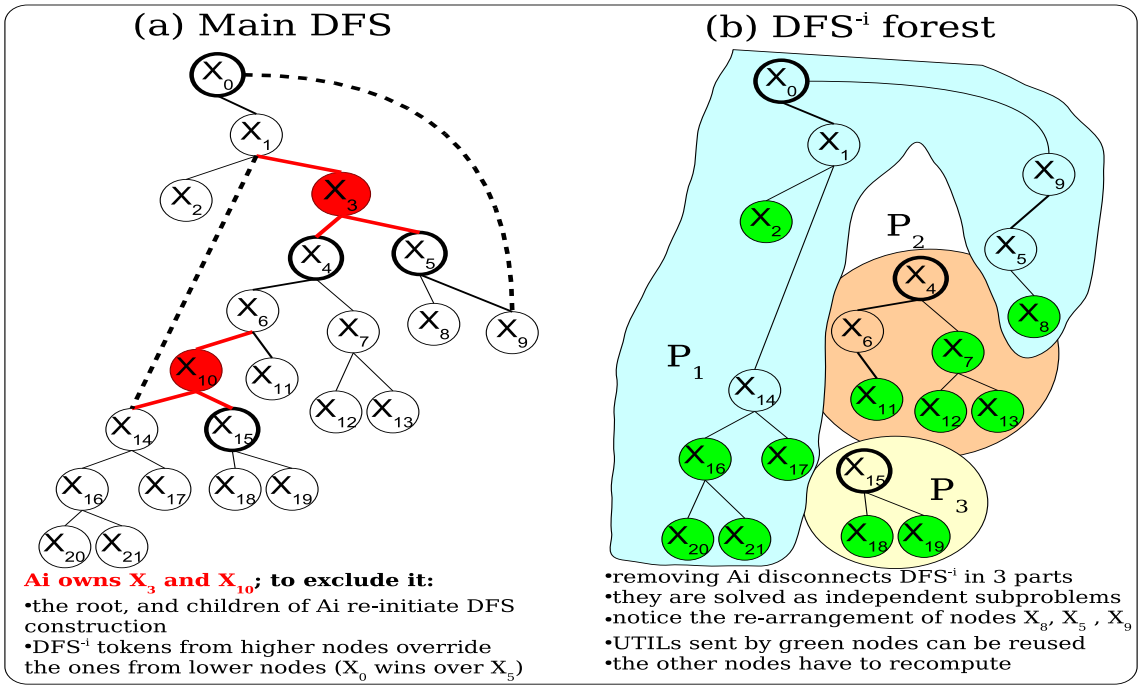


Figure 11.4: Reconstructing $DFS(-A_i)$ from $DFS(\mathcal{A})$ in M-DPOP. The result is in general a DFS forest. The bold nodes from main DFS initiate DFS^{-i} propagation. The one initiated by X_5 is redundant and eventually stopped by X_9. The ones from X_4 and X_{15} are useful, as their subtrees become really disconnected after removing A_i. X_{14} does not initiate any propagation since it has X_1 as a pseudoparent. X_1 is not controlled by A_i, and will eventually connect to X_{14}. Notice that $X_0 - X_9$ and $X_1 - X_{14}$ are turned into tree edges.

agent A_i controls only X_3 and X_{10}. Then it has no way of influencing the messages sent in the subtrees rooted at $\{X_{14}, X_{15}, X_2, X_7, X_5, X_{11}\}$. We want to be able to reuse as many of these UTIL messages as possible. In solving the problem with agent A_i removed we will strive to construct a DFS^{-i} arrangement for problem $DCOP(-A_i)$ that is as similar as possible to the DFS for the main problem. This is done with the goal of maximizing the re-use of computation across problems. See Figure 11.4(b). Notice that this is now a DFS forest, with three distinct connected components. The UTIL messages that were sent by the green nodes can be re-used in solving $DCOP(-A_i)$. These are all the UTIL messages sent by nodes in the subtrees that were not influenced by agent A_i except for $\{X_{14}, X_{15}, X_5\}$ and also X_9, which now has a different local DFS arrangement.

M-DPOP uses the "safe reusability" idea suggested by this example. See Algorithm 26. In its first stage, M-DPOP solves the main problem just as in Simple-M-DPOP. Once this is complete, each marginal problem $DCOP(-A_i)$ is solved in parallel. To solve $DCOP(-A_i)$, a DFS^{-i} forest (it will be a forest in the case that $DCOP(-A_i)$ becomes disconnected) is constructed as a modification to $DFS(\mathcal{A})$, retaining as much of the structure of $DFS(\mathcal{A})$ as possible. A new $DPOP(-A_i)$ execution is performed on the DFS^{-i} and $UTIL$ messages are determined to be either *reusable* or *not reusable* by the sender of the message based on the differences between DFS^{-i} and $DFS(\mathcal{A})$. We will explain

Algorithm 26 *M-DPOP: faithfully reuses computation from the main problem.*

Run DPOP for $DCOP(\mathcal{A})$ on $DFS(\mathcal{A})$; find X^*
forall $A_i \in \mathcal{A}$ **do**
  in parallel
Create $\boldsymbol{DFS^{-i}}$ with Algorithm 27 by adjusting $DFS(\mathcal{A})$
Run DPOP for $\boldsymbol{DCOP(-A_i)}$ **on** $\boldsymbol{DFS^{-i}}$:
if *leaves in* DFS^{-i} *observe no changes in their* DFS^{-i} **then**
  they send $null$ $UTIL^{-i}$ messages
else they compute their $UTIL^{-i}$ messages anew, as in DPOP
subsequently, all nodes $X_k \in DFS^{-i}$ do:
if X_k *receives only* $null$ $UTIL^{-i}$ *msgs* $\wedge$ *(*$P_k = P_k^{-i} \wedge PP_k = PP_k^{-i} \wedge C_k = C_k^{-i}$*)* **then**
  X_k sends a $null$ $UTIL^{-i}$ message
else
  node X_k computes its $UTIL^{-i}$ message, **reusing**:
  forall $X_l \in Neighbors(X_k)$ *s.t.* X_l *sent* $UTIL^{-i} = null$ **do**
    X_k **reuses** the $UTIL$ message X_l had sent in $DCOP(\mathcal{A})$
Compute and levy taxes as in simple-M-DPOP;
Each A_i assigns values in X^* as the solution to its local COP_i;

below how DFS^{-i} is constructed.

11.5.1 Phase One of M-DPOP for a Marginal Problem: Constructing DFS^{-i}

Given a graph $DCOP(\mathcal{A})$ and a DFS arrangement $DFS(\mathcal{A})$ of $DCOP(\mathcal{A})$, if one removes a set of nodes $X(A_i) \in DCOP(\mathcal{A})$ (the ones that belong to A_i), then we need an algorithm that constructs a DFS arrangement, DFS^{-i}, for $DCOP(\mathcal{A}) \setminus X(A_i)$. We want to achieve the following properties:

1. DFS^{-i} must represent a correct DFS arrangement for the graph $DCOP(-A_i)$ (a DFS forest in the case $DCOP(-A_i)$ becomes disconnected).
2. DFS^{-i} must be constructed in a way that is non-manipulable by A_i, i.e. without allowing agent A_i to interfere with its construction.
3. DFS^{-i} should be as similar as possible to $DFS(\mathcal{A})$. This allows for reusing *UTIL* messages from $DPOP(\mathcal{A})$, and saves on computation and communication.

The main difficulty stems from the fact that removing the nodes that represent variables of interest to agent A_i from $DFS(\mathcal{A})$ can create disconnected subtrees. We need to reconnect and possibly rearrange the (now disconnected) subtrees of $DFS(\mathcal{A})$ whenever this is possible. Return to the example in Figure 11.4. Removing agent A_i and nodes X_3 and X_{10} disrupts the tree in two ways: some subtrees become completely disconnected from the rest of the problem (e.g. $X_{15} - X_{18} - X_{19}$); some other ones

Algorithm 27 *Reconstruction of DFS^{-i} from $DFS(\mathcal{A})$. All data structures for the DFS^{-i} are denoted with superscript $^{-i}$.*

Procedure Token_passing for DFS^{-i} (executed by all nodes $X_k \notin X(A_i)$) :

forall $X_l \in Neighbors(X_k)$ *s.t.* X_l *belongs to* A_i **do**

1 Remove X_l from $Neighbors(X_k)$ and from C_k, PC_k, PP_k *//i.e. send nothing to* A_i

2 Sort $Neighbors(X_k)$ in this order: C_k, PC_k, PP_k, P_k *//mimic* $DFS(\mathcal{A})$
if X_k *is root, or* $P_k \in X(A_i)$ *(i.e. executed by the root and children of* A_i*)* **then**

3 Initiate DFS^{-i} as in normal DFS (Algorithm 24)

4 **else do** Process_incoming_tokens()

5 Send $DFS^{-i}(X_k)$ back to P_k^{-i} *// X_k's subtree completely explored*

Procedure Process_incoming_tokens()

6 Wait for any incoming DFS^{-i} token; Let X_l be its sender

7 **if** $X_l \in A_i$ **then** ignore message

8 **else**

9 **if** *this is first token received* **then**

10 $P_k^{-i} = X_l$; $PP_k^{-i} = \{X_j \neq P_k^{-i} | X_j \in Neighbors(X_i) \cap DFS^{-i}\}$

11 $root_k^{-i}$= first node in the token DFS^{-i}

else

12 let X_r be the first node in DFS^{-i}

13 **if** $X_r \neq root_k^{-i}$ *//i.e. this is another* DFS^{-i} *traversal* **then**

14 **if** *depth of* X_r *in* $DFS(\mathcal{A}) <$ *depth of* $root_k^{-i}$ *in* $DFS(\mathcal{A})$ **then**

15 Reset $P_k^{-i}, PP_k^{-i}, C_k^{-i}, PC_k^{-i}$ *//override redundant DFS from lower root*

16 $P_k^{-i} = X_l$; $PP_k^{-i} = \{X_j \neq P_k^{-i} | X_j \in Neighbors(X_i) \cap DFS^{-i}\}$

17 $root_k^{-i} = X_r$

18 Continue as in Algorithm 24

remain connected only via back-edges, thus forming an invalid DFS arrangement (e.g. $X_5 - X_8 - X_9$). The basic principle we use is to reconnect disconnected parts via back-edges from $DFS(\mathcal{A})$ whenever possible. This is intended to preserve as much of the structure of as possible. For example, in Figure 11.4, the back edge $X_0 - X_9$ is turned into a tree edge, and X_5 becomes X_9's child. Node X_8 remains X_5's child.

The DFS^{-i} reconstruction algorithm is presented in Algorithm 27. The high-level overview is as follows (in bold we state the purpose of each step):

1. (**Similarity to $DFS(\mathcal{A})$:**) All nodes retain the DFS data structures from constructing $DFS(\mathcal{A})$; i.e., the lists of their children, pseudo parents/children, and their parents from $DFS(\mathcal{A})$. They

will use this data as a starting point for building the DFS arrangements, $DFS(-A_i)$, for marginal problems.

2. (**At least one traversal of each connected component on a DFS forest:**) The root of $DFS(\mathcal{A})$ *and the children*[11] *of removed nodes* each initiate a DFS^{-i} token passing as in $DFS(\mathcal{A})$, except for these changes:

   - Each node X_k sends the token only to neighbors not owned by A_i.
   - The order in which X_k sends the token to its neighbors is based on $DFS(\mathcal{A})$: First X_k's children from $DFS(\mathcal{A})$, then its pseudochildren, then its pseudoparents, and then its parent. This order helps preserve structure from $DFS(\mathcal{A})$ into $DFS(-A_i)$.

3. (**Unique traversal of each connected component on a DFS forest:**) Each node X_k retains its "root path" in $DFS(\mathcal{A})$ and knows its depth in the DFS arrangement. When a new token DFS^{-i} arrives:

   - If it is the first DFS^{-i} token that arrives, then the sender (let this be X_l) is marked as the parent of X_k in DFS^{-i}: $P_k^{-i} = X_l$. Notice that X_l could be different from the parent of X_k from $DFS(\mathcal{A})$. X_k stores the first node from the received token DFS^{-i}, as $root_k^{-i}$: the (provisional) root of the connected component to which X_k belongs in $DCOP(-A_i)$.
   - If this is not the first DFS^{-i} token that arrives, then there are two possibilities:
     - the token received is part of the same DFS^{-i} traversal process. X_k recognizes this by the fact that the first node in the newly received token is the same as the previously stored $root_k^{-i}$. In this case, X_k proceeds as normal, as in Algorithm 24: marks the sender as pseudochild, etc.
     - the token received is part of *another* DFS^{-i} traversal process, initiated by another node than $root_k^{-i}$ (see below in text for when this could happen). Let X_r be the first node in the newly received token. X_k recognizes this situation by the fact that X_r is **not** the same as the previously stored $root_k^{-i}$. In this case, the DFS^{-i} traversal initiated by the higher node in $DFS(\mathcal{A})$ prevails, and the other one is dropped. To determine which traversal to pursue and which one to drop, X_k compares the depths of $root_k^{-i}$ and X_r in $DFS(\mathcal{A})$. If X_r is higher, then it becomes the new $root_k^{-i}$. X_k overrides all the previous DFS^{-i} information with the one from the new token. It then continues the token passing with the new token as in Algorithm 24.

To see why it is necessary to also start propagations from the children of removed nodes (step 2), consider again the example from Figure 11.4. Removing X_{10} and X_3 completely disconnects the subtree $\{X_4, X_6, X_{11}, X_7, X_{12}, X_{13}\}$. Had X_4 not started a propagation, this subtree would not have

[11] Children which have pseudoparents above the excluded node, for instance X_{14} in Figure 11.4, do not initiate DFS token passing because it would be redundant: they would eventually receive a DFS token from their pseudoparent.

been visited at all since there are no connections between the rest of the problem and any nodes in the subtree.[12]

Lemma 1 (DFS correctness) *Algorithm 27 constructs a correct DFS arrangement (or forest),* DFS^{-i} *for* $DCOP(-A_i)$ *given a correct DFS arrangement* $DFS(\mathcal{A})$ *for* $DCOP(\mathcal{A})$.

PROOF. First, since a DFS^{-i} is started from each child of a node that was controlled by A_i, and also from the root, it is ensured that each connected component is DFS-traversed at least once (follows from Step 2). Second, each DFS process is similar to a normal DFS construction, in that each node sends the token to all its neighbors (except for the ones controlled by A_i); it is just that they do so in a pre-specified order (the one given by $DFS(\mathcal{A})$). It follows that all nodes in a connected component will eventually be visited (follows from Step 3). Third, higher-priority DFS traversals override the lower priority ones (i.e. DFS traversals initiated by nodes higher in the tree have priority), again by Step 3. Eventually one single DFS-traversal is performed in a single connected component. □

Lemma 2 (DFS robustness) *The DFS arrangement,* DFS^{-i}*, constructed by Algorithm 27 is non-manipulable by agent* A_i*, for any input DFS arrangement from the solution phase to* $DCOP(\mathcal{A})$.

PROOF. This follows directly from Step 3, since A_i does not participate in the process at all: its neighbors do not send it any messages (see Algorithm 27, line 1), and any messages it may send are simply ignored (see Algorithm 27, line 7) □

In fact, **no** additional links are created while constructing DFS^{-i}. The only possible changes are that some edges can reverse their direction (parents/children or pseudoparents-pseudochildren can switch places), and existing back-edges can turn into tree edges. Again, one can see this in Figure 11.4.[13]

11.5.2 Phase Two of M-DPOP for a Marginal Problem: $UTIL^{-i}$ propagations

Once DFS^{-i} is built, the marginal problem without A_i is then solved on DFS^{-i}. Utility propagation proceeds as in normal DPOP except that nodes determine whether the *UTIL* message that was sent in

[12] Some of the DFS traversals initiated in Step 2 are redundant and the same part of the problem graph can be visited more than once. The simple overriding rule in Step 3 ensures that only a single DFS^{-i} tree is eventually adopted in each connected component, namely the one that is initiated by the *highest node* in the *original* $DFS(\mathcal{A})$. For example, in Figure 11.4, X_5 starts an unnecessary DFS^{-i} propagation, which is eventually stopped by X_9, which receives a higher priority DFS^{-i} token from X_0. Since X_9 knows that X_0 is higher in $DFS(\mathcal{A})$ than X_5, it drops the propagation initiated by X_5, and relays only the one initiated by X_0. It does so by sending X_5 the token for DFS^{-i} received from X_0 to which it adds itself. Upon receiving the new token from X_9, node X_5 realizes that X_9 is its new parent in DFS^{-i}. Thus, the redundant propagation initiated by X_5 is eliminated and the result is a consistent DFS subtree for the single connected component P_1.

[13] A simple alternative is to have children of all nodes X_k^i that belong to A_i, create a bypass link to the first ancestor of X_k^i that does not belong to A_i. For example, in Figure 11.4, X_4 and X_5 could each create a link with X_1 to bypass X_3 completely in $DFS(-A_i)$. However, additional communication links may be required in this approach.

$DPOP(\mathcal{A})$ can be reused. This is signaled to their parent by sending a special *null UTIL* message. More specifically, the process is as follows:

- The leaves in DFS^{-i} initiate $UTIL^{-i}$ propagations:
  1. If the leaves in DFS^{-i} observe no changes in their local DFS^{-i} arrangement as compared to $DFS(\mathcal{A})$ then the *UTIL* message they sent in $DCOP(\mathcal{A})$ remains valid and they announce this to their parents by sending instead a $null\ UTIL^{-i}$ message.
  2. Otherwise, a leaf node computes its *UTIL* message anew and sends it to their (new) parent in DFS^{-i}.
- All other nodes wait for incoming $UTIL^{-i}$ messages and:
  1. If *every* incoming messages a node X_k receives from its children is $null$ *and* there are no changes in the parent/pseudoparents then it can propagate a $null\ UTIL^{-i}$ message to its parent.
  2. Otherwise, X_k has to recompute its $UTIL^{-i}$ message. It does so by reusing all the *UTIL* messages that it received in $DCOP(\mathcal{A})$ from children that have sent it $null$ messages in $DCOP(-A_i)$ and joining these with any new *UTIL* messages received.

Example 26 (Reusing Computation) *Consider $DCOP(-A_i)$ in Figure 11.4, where X_{16} and X_{17} are children of X_{14}. X_{14} has to recompute a UTIL message and send it to its new parent X_1. To do this, it can reuse the messages sent by X_{16} and X_{17} in $DCOP(\mathcal{A})$, because both sending subtrees do not contain A_i. By doing so, X_{14} reuses the effort spent in $DCOP(\mathcal{A})$ to compute the messages $UTIL_{20}^{16}$, $UTIL_{21}^{16}$, $UTIL_{16}^{14}$ and $UTIL_{17}^{14}$.*

Theorem 10 *The M-DPOP algorithm is a faithful distributed implementation of efficient social choice and terminates with the outcome of the VCG mechanism.*

PROOF. From the partition principle. First, agent A_i cannot prevent the construction of a valid DFS^{-i} for $DCOP(-A_i)$ (Lemmas 1 and 2). Second, agent A_i cannot influence the execution of DPOP on $DCOP(-A_i)$ because all messages that A_i influenced in the main problem $DCOP(\mathcal{A})$ are recomputed by the system without A_i. The rest of the proof follows as for simple-M-DPOP, leveraging the locality of the tax payment messages and the enforcement provided by the bank and via the catastrophic failure assumption. □

11.5.3 Experimental Evaluation: Distributed Meeting Scheduling

We present the results of our experimental evaluation of DPOP, Simple M-DPOP and M-DPOP in a distributed meeting scheduling problem. The problems consist of agents working for a large organization and representing individuals, or groups of individuals, for the purpose of scheduling meetings for

some upcoming period of time. Although the agents themselves are self interested, the organization as a whole requires an optimal overall schedule, that minimizes cost (alternatively, maximizes the utility of the agents). This makes it necessary to use a faithful distributed implementation such as M-DPOP. In enabling this, we can imagine that the organization distributes a virtual currency to each agent (perhaps using this to prioritize particular participants.)

The problem is modeled as a DCOP as described in Section 2.3.1, with each agent assigning a utility to each possible time for each meeting by imposing a unary relation on each variable X_j^i. Each such relation is private to A_i, and denotes how much utility A_i associates with starting meeting M_j at each time $t' \in d_j$, where d_j is the domain for meeting M_j. The social objective is to find a schedule in which the total utility is maximized while satisfying the all-different constraints for each agent.[14]

Following[127], we model the organization by providing a *hierarchical structure*. In a realistic organization, the majority of interactions are within departments, and only a small number are across departments and even then these interactions will typically take place between two departments adjacent in the hierarchy. This hierarchical organization provides structure to our test instances: with high probability (around 70%) we generate meetings within departments, and with a lower probability (around 30%) we generate meetings between agents belonging to parent-child departments. We generated random problems having this structure,[15] with an increasing number of agents: from 10 to 100 agents. Each agent participates in 1 to 5 meetings, and has a uniform random utility between 0 and 10 for each possible schedule for each meeting in which it participates. The problems are generated such that they have feasible solutions.

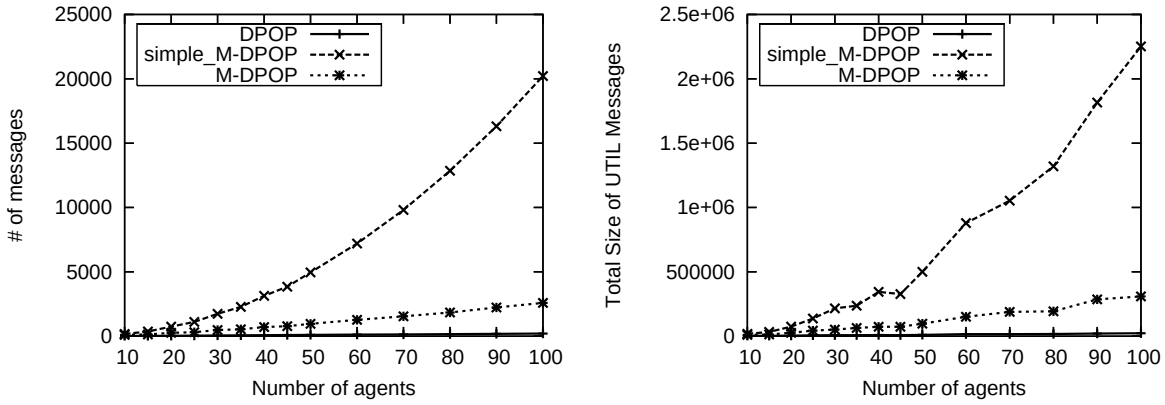
For each problem size, we averaged the results over 100 different instances. We solved the main problems using DPOP and the marginal ones using simple-M-DPOP, and M-DPOP respectively. All experiments were performed in the FRODO multiagent simulation environment[154], on a 1.6Ghz/1GB RAM laptop. FRODO is a simulated multiagent system, where each agent executes asynchronously in its own thread, and communicates with its peers only via message exchange.

These experiments were geared towards showing how much effort M-DPOP is able to reuse from the main to the marginal problems. Figure 11.5 shows the absolute computational effort in terms of number of messages (Figure 11.5(a)), and in terms of the total size of the messages exchanged, in bytes (Figure 11.5(b)). The curves for DPOP represent just the number of messages (total size of messages, respectively) required for solving the main problems, and not also the marginal ones. The curves for simple-M-DPOP and M-DPOP represent the total number (size, respectively) of *UTIL* messages, for both main and marginal problems.

We notice several interesting facts. First, the number of messages required by DPOP increases

[14] In a simple variation one could also seek to maximize the weighted utility across the agents, wherein some agents receive more priority within the organization than other agents. The VCG payments, and also M-DPOP, can be easily extended to provide appropriate incentives in this setting.

[15] The test instances can be found at http://liawww.epfl.ch/People/apetcu/research/mdpop/MSexperiments.tgz



(a) Number of messages

(b) Total size of *UTIL* messages (in valuations)

Figure 11.5: Meeting scheduling problem: measures of absolute computational effort (in terms of the number of messages sent and the total size of the *UTIL* messages) in DPOP, simple-M-DPOP and M-DPOP. The curves for DPOP represent effort spent just on the main problem, while the ones for simple-M-DPOP and M-DPOP represent effort on both the main and the marginal problems.

linearly with the number of agents because DPOP's complexity in terms of number of messages is always linear in the size of the problem. On the other hand, the number of messages of simple-M-DPOP increases roughly quadratically with the number of agents, since it solves a linear number of marginal problems from scratch using DPOP, each requiring a linear number of messages. The performance of M-DPOP lies somewhere between the DPOP and simple-M-DPOP with more advantage achieved over simple-M-DPOP as the size of the problem increases, culminating with almost an order of magnitude improvement for the largest problem sizes (i.e. with 100 agents in the problem). Similar observations can be made about the total size of the *UTIL* messages (a good measure of computation, traffic and memory requirements) by inspecting Figure 11.5(b). For both metrics we find that the performance of M-DPOP is only slightly super-linear in the size of the problem.

Figure 11.6 shows the percentage of the additional effort required for solving the marginal problems that can be reused from the main problem, i.e. the probability that a *UTIL* message required in solving a marginal problem can be taken directly from the message already used in the main problem. We clearly see that as the problem size increases we can actually reuse more and more computation from the main problem. The intuition behind this is that in large problems, each individual agent is localized in a particular area of the problem. This translates into the agent being localized in a specific branch of the tree, thus rendering all computation performed in other branches reusable for the marginal problem that corresponds to that respective agent. Looking also at the percentage of reuse when defined in terms of message size rather than the number of messages we see that this is also trending upwards as the size of the problem increases.

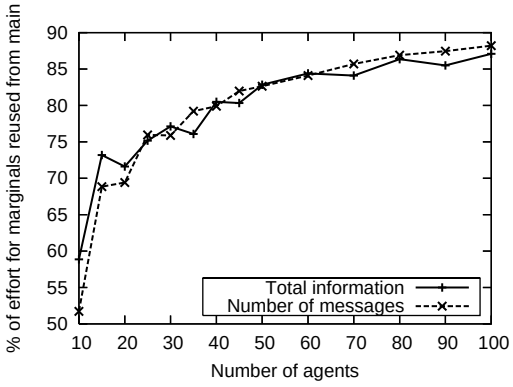


Figure 11.6: Meeting scheduling problem: Percentage of effort required for the marginal problems that is reused by M-DPOP from the main problem. Reuse is measured both in terms of the percentage of the *UTIL* messages that can be reused (dashed) and also in terms of the total size of the *UTIL* messages that are reused as a fraction of the total *UTIL* message size (solid).

11.5.4 Summary of M-DPOP

M-DPOP is a faithful, distributed algorithm with which one can solve efficient social choice problems in multi-agent systems with private information and agent self-interest. No agent can improve its utility either by misreporting its local information or deviating from any aspect of the algorithm (e.g., computation, message-passing, information revelation.) The only centralized control we assume is that of a bank that is able to receive messages about payments and collect payments. In addition to promoting efficient decisions we also minimize the amount of additional computational effort required for computing the VCG payments by reusing effort from the main problem. Experimental results show that a significant amount of the computation required in all the main problems can be reused from the main problem, sometimes above 87%. This provides near-linear scalability in massive, distributed social choice problems that have local structure so that the maximal induced tree width is small.

An issue for future work relates to robustness against *adversarial* or *faulty* agents: the current solution is fragile in this sense, with its equilibrium properties relying on other agents following the protocol. Some papers[4, 124, 191] provide robustness to mixture models (e.g. some rational, some adversarial) but we are not aware of any work with these mixture models in the context of efficient social choice. Another interesting direction is to find ways to allow for approximate social choice (e.g. with memory-limited DPOP variations[158]) while retaining incentive properties, perhaps in approximate equilibria. Future research should also consider the design of distributed protocols that are robust against false-name manipulations in which agents can participate under multiple pseudonyms[229], and achieve better robustness through mitigating opportunities for collusive behavior and removing weak equilibria in favor of strict equilibria[5, 108].

11.6 Achieving Faithfulness with other DCOP Algorithms

The partition principle, described in Section 11.4.3, is algorithm independent. The question as to whether another, optimal DCOP algorithm can be made faithful therefore revolves, critically, around whether the algorithm will satisfy the robustness requirement of the partition priciple. We make the following observations:

- Robustness in the first sense, i.e. that no agent A_i can influence the solution to the efficient SCP without agent A_i, is always achievable at the cost of restarting computation on the marginal problem with each agent removed in turn, just as we proposed for simple-M-DPOP.

- Robustness in the second sense, i.e. that no agent A_i can influence the report(s) that the bank receives about the negative externality that A_i imposes on the rest of the system, conditioning on the solutions to the main problem and the problem without A_i, is also immediate because of the locality property of tax payments, and as long as the DCOP algorithm terminates with every agent knowing the part of the solution that is relevant in defining its own utility.

Thus, if one is content to restart the DCOP algorithm multiple times, then the same kinds of results that we provide for simple-M-DPOP are generally available. This is possible because of the already mentioned locality property of payments, which follows from the disaggregation of the VCG payment across agents in Eq. (11.9) and because of the information and communication structure of DCOP. The other useful property of DCOP in this context, worth reemphasizing, is that it is possible to retain faithfulness even when one agent plays a *pivotal role* in connecting the problem graph. Suppose that problem, $DCOP(-A_i)$, becomes disconnected without A_i. But, if this is the case then its optimal solution is represented by the union of the optimal solution to each connected subcomponent of the problem, and no information needs to flow between disconnected components either for the purpose of solving the problem or for the purpose of reporting the components of agent A_i's tax.

We discuss in the following possible adaptations of the other two most popular algorithms for DCOP: ADOPT and OptAPO.

11.6.1 Adapting ADOPT for Faithful, Efficient Social Choice

ADOPT (reviewed in Chapter 3) is one of the most celebrated algorithms for DCOP. Considering its main advantage of requiring only polinomial memory, it seems legitimate to ask the question: *"Could ADOPT be used for faithfully solving the SCP"*. We discuss this possibility in the next two sections.

11.6.1.1 Adaptation of ADOPT to the DCOP model with replicated variables

ADOPT's complexity is given by the number of messages, which is exponential in the height of the DFS tree. Similar to DPOP, using the DCOP model with replicated variables could artificially increase the complexity of the solving process. Specifically, the height of the DFS tree is increased when using replicated variables compared to the centralized problem graph.

ADOPT can be modified to exploit the special structure of these replicated local variables in a similar way as DPOP. Specifically, ADOPT should explore sequentially only the values of the original variable, and ignore assignments where replicas of the same variable take different values. This works by allowing just the agent that owns the highest replica of each variable to freely choose values for the variable. This agent then announces the new value of the variable to all other agents owning replicas of the variable. These other agents would then consider just the announced value for their replicas, add their own corresponding utilities, and continue the search process. Using this special handling of the replica variables, the resulting complexity is no longer exponential in the height of the distributed DFS tree, but in the height of the DFS tree obtained by traversing the original problem graph.

For example, in Figure 11.1, it is sufficient to explore the values of M_3^2, and directly assign these values to M_3^3 and M_3^1 via *VALUE* messages, without trying all the combinations of their values. This reduces ADOPT's complexity from exponential in 6, to exponential in 3.

11.6.1.2 Reusability of computation in ADOPT

Turning to re-use of computation, we note that because ADOPT uses a DFS arrangement then it is easy to identify which parts of the DFS arrangement for the main problem are impossible for an agent to manipulate, and therefore can be "reused" while computing the solution to the marginal problem with that agent removed. Just as with DPOP, the DFS reconstruction techniques from Section 11.5.1 apply.

However, a major difference between DPOP and ADOPT is that in DPOP, each agent stores its outgoing *UTIL* message, and thus has available all the utilities contingent to all assignments of the variables in the agent's separator. This makes it possible for the agent to simply reuse that information in all marginal problems where the structure of the DFS proves it is safe to do so. In contrast, ADOPT does not store all this information because of its linear memory policy. This in turn makes it impossible to reuse computation as in DPOP from the main problem to the marginal problems. All marginal problems have to be solved from scratch, and thus the performance would scale poorly as problem size increases.

We see two alternatives for addressing this problem: (a) renounce linear memory guarantees, and use a caching scheme like in NCBB[32] or dAOBB(i)[170]: this would allow for a similar reusability as in M-DPOP, where previously computated utilities can be extracted from the cache instead of having to be recomputed. Alternatively, (b) one can devise a scheme where the previously computed best

solution can be saved as a reference, and subsequently used as an approximation while solving the marginal problems. This could possibly provide better bounds and thus allow for better pruning, such that some computation could be saved. Both these alternatives are outside the scope of this thesis, and considered for future work.

11.6.2 Adapting OptAPO for Faithful, Efficient Social Choice

OptAPO (reviewed in Chapter 3) is the other most popular algorithm for DCOP. Similar to the adaptations of DPOP and ADOPT to social choice, OptAPO can also be made to take advantage of the special features of the DCOP model with replicated variables. Its complexity then would not be artificially increased by the use of this DCOP model.

OptAPO has the particularity that it uses mediator agents to *centralize subproblems* and solve them in dynamic and asynchronous mediation sessions. The mediator agents then announce their results to the other agents, who have previously sent their subproblems to the mediators. This process alone would introduce additional possibility for manipulation in a setting with self interested agents. However, using the VCG mechanism would fix this problem and incentivise the agents to behave correctly according to the protocol.

As with ADOPT, the main issue with using OptAPO for faithful social choice is the reusability of computation from the main to the marginal problems. Specifically, consider that while solving the main problem, a mediator agent A_i has centralized and aggregated the preferences of a number of other agents, while solving subproblems as dictated by the OptAPO protocol. Subsequently, when trying to compute the solution to the marginal problem without agent A_i, all this computation has to go to waste, as it could have been manipulated by A_i while solving the main problem.

Furthermore, since OptAPO does not explicitely use structure in the problem, it is unclear whether *any* computation from the main problem could be safely reused in any of the marginal problems. To make matters worse, experimentaly studies ([44, 169]) show that in many situations, OptAPO ends up relying on a single agent in the system to centralize and solve the whole problem. This implies that while solving the marginal problem without that agent, one can reuse zero effort from the main problem.

Chapter 12

Budget Balance

For social choice problems with self interested agents, the VCG mechanism achieves efficiency, individual-rationality and incentive-compatibility. One of the characteristics of the VCG mechanism is that it requires wasting the taxes collected from the agents, thus decreasing their net utility. Burning money in this way can be a particular problem in networked systems where payments are made by "proof of work"[61] or other form and the primary goal is social efficiency, not revenue.

This chapter introduces two extensions to M-DPOP (Chapter 11) that address this problem of burning money. Our extensions exploit structure in the problem to develop faithful methods to redistribute payments back to agents, reducing this cost on the system. The first method (R-M-DPOP) preserves the efficiency guarantees, but cannot guarantee full budget balance (some taxes may still have to be wasted). Nevertheless, our experimental results show that we can redistribute a significant percentage of the VCG taxes (up to 70% in our experiments). The second redistribution scheme (BB-M-DPOP) guarantees complete budget balance, but cannot guarantee optimality. BB-M-DPOP works by forcibly limiting each agent's influence to a restricted area, which in turn allows for an effective redistribution of all of the VCG payments in a faithful way. Interestingly, BB-M-DPOP yields better net utility for the system as a whole, even though it does not guarantee optimal solutions. In our experiments, BB-M-DPOP, R-M-DPOP and VCG-classic provided agents with a net utility of 97%, 89%, and 71% from the cooperative optimum, respectively.

We have seen in the previous chapter that distributed optimization problems can model social choice problems with self interested agents. We have introduced the M-DPOP algorithm, which is the first faithful distributed algorithm for general social choice problems that deals with self-interested users. M-DPOP implements the Vickrey-Clarke-Groves mechanism (VCG) [37, 91, 213], which aligns the incentives of the participating agents with the goal of maximizing overall utility. VCG frees agents of the burden of reasoning strategically about their actions, and makes honest behaviour a dominant strategy equilibrium.

In the VCG mechanism, each agent makes a payment that equals the negative marginal externality that its presence imposes on the rest of the system, in terms of influencing the ultimate choice of values

for variables. Sometimes these payments are large, and thus decrease the agents' *net* utility significantly (recall from Chapter 11 that net utility means the utility derived from the optimal solution, minus the VCG payments). These payments cannot be simply returned to the agents, because this would break the incentive properties. Agents would falsely declare their preferences such that they get tax refunds. For instance, an agent might try to increase the tax payments made by other agents by overstating the negative impact of those agents on its own local solution in order to increase the payments made by these agents and in turn its own share of these payments.

The simplest way to deal with this problem is to just waste all payments ("burn" the money, or give it to some external third party). However, this approach creates a loss in the overall net utility of the system, and possibly creates unwanted incentives for a third party receiving the payments. Burning money in this way can be a particular problem in netwoked systems where payments are made by "proof of work"[61], and in which the primary goal is efficiency and the revenue accrues to no-one and payments are (literally) wasted compute cycles.

Fundamental results in mechanism design prove the *impossibility* of a *general* mechanism that satisfies at the same time optimality [1], individual rationality [2], incentive-compatibility [3] and budget-balance [4] in a dominant strategy equilibrium[101]. [5] Hence, at least one of these properties has to be violated. In this chapter, we design new mechanisms that retain a dominant-strategy equilibrium,[6] and sacrifice either efficiency or budget-balance. We seek to modify the VCG mechanism, by redistributing the payments proposed by the mechanism back to agents, but in a way that does not compromise the incentive properties. In one method, this is achieved by imposing *ex ante* constraints on the optimization problem, which has the effect of redistributing the payments of a modified VCG mechanism, that is in effect applied to this additionally constrained problem. These constraints are actually *unconstraining*, in the sense that they are introduced to provide additional flexibility in redistributing payments.

Our results are presented as extensions to M-DPOP, which was introduced in Chapter 11. The first method (R-M-DPOP) guarantees efficiency, but does not guarantee full redistribution and thus is not exactly budget balanced. We note however that it *never* runs at a deficit: the bank always receives a non-negative amount of payments from the agents. Moreover, this method is typically able to redistribute a considerable amount of the payments proposed by the VCG mechanism. The second method (BB-M-DPOP) offers the inverse tradeoff: it guarantees *exact* budget balance, but sometimes at the expense of efficiency. Both these methods exploit problem structure, albeit in quite different ways. R-M-DPOP

[1] Also called "economic efficiency", it means that the optimal solution to the social choice problem must be chosen.

[2] Also called "participation constraint" and means that no agent should be charged more than the utility it derives from the decision.

[3] Each agent's utility is maximized when truthfully declaring its preferences

[4] Sum of payments from agents to any third party must equal zero.

[5] Myerson and Satterthwaite[144] also establish that it is impossible to satisfy efficiency, budget-balance and individual-rationality even in a Bayes-Nash equilibrium.

[6] The equilibrium concept will be ex post Nash when used as a distributed implementation, but our mechanisms provide a dominant-strategy equilibrium when they are used as centralized mechanisms and computation and message-passing is not passed to agents.

identifies components of the problem that define payments that cannot be influenced by some subset of agents, so that these agents are then eligible to receive a share of the payments. BB-M-DPOP places *ex ante* constraints on the problem, forcibly preventing each agent from having any influence on decisions for some part of the domain. This also creates, for each agent, payments made by other agents that cannot be influenced by the agent. (These are the payments of the agents that care only about the problem domain that cannot be influenced by this agent.) BB-M-DPOP can leverage problem structure to decide how to constrain the problem, with constraints added where the likely influence of an agent is very weak.

In achieving our results we propose a "label propagation" algorithm, that is used in R-M-DPOP to determine the subset of variables that an agent could not have *possibly* influenced, for all its possible reports, given the reports of the others. Such an agent could then receive, if elected as a candidate for this redistribution, the VCG payment made by another agent that is only negatively impacted in its utility by decisions made in regard to these variables. In BB-M-DPOP, we propose a configurable method that allows each agent to express its preferences on its variables of interest, and even indirectly influence other variables via other agents' relations. This method allows nevertheless the influence of each agent to be decisevely cut beyond a configurable point, such that the redistribution of taxes originating beyond the given point is not influenceable by the agent in question. This works by propagating dual UTIL messages, corresponding to both the main problem (the influence of the agent in question included), and to the marginal problem (the influence removed). Beyond the cutoff point, only the marginal message is propagated, which effectively eliminates any influence from the agent.

As a distributed implementation, both R-M-DPOP and BB-M-DPOP retain faithfulness when coupled with the centralization of the pre-processing step in which a depth first search (DFS) arrangement is constructed for the problem graph. This is performed by the agents themselves in M-DPOP, but the DFS arrangement is used by R-M-DPOP and BB-M-DPOP in determining *whether* and *to whom* to redistribute payments; therefore, the agents have vested interest in manipulating the DFS creation which in turn would influence the redistribution schemes (see Section 12.4.1). The centralization of this pre-processing step can be achieved without a third party needing to know about the private information of agents; e.g. their private variables or local utility information: all is needed is access to the public information about each agent's variables of possible interest. One natural party to perform this task in our new methods could be the *bank*, that is a trusted third party required by M-DPOP in order to enforce the collection of payments. Indeed, it is natural to think the bank should play a more active role in BB-M-DPOP and R-M-DPOP given that these are methods to allow for the incentive-compatible redistribution of payments.

R-M-DPOP and BB-M-DPOP can also be used as centralized algorithms, in which the agents report their private information to a "center" as in traditional mechanism design. The center would then directly implement R-M-DPOP, or BB-M-DPOP, which correspond as centralized algorithms to dynamic programming with generalized bucket elimination[51], coupled here with checking for the possible influence of an agent on parts of the problem domain. As a centralized mechanism, R-M-

DPOP can be viewed as extending, and exemplifying, the principles of Cavallo[29], in that we leverage structure on agent valuations and determine *given reports of other agents*, which parts of the problem domain can be influenced by an agent. BB-M-DPOP, on the other hand, can be viewed as generalizing the method of Faltings[68], who proposed to constrain social choice problems in that a single agent is prevented from being able to have any influence on the *entire* problem, and can therefore receive the payments made by all the other agents.

We present experimental results, in a simulated meeting scheduling domain, that show that R-M-DPOP can redistribute a significant percentage of the VCG payments (as much as around 70%). R-M-DPOP does this while retaining perfect efficiency, i.e. while optimally solving the social choice problem. Interestingly, BB-M-DPOP yields better net utility for the system as a whole, even though it does not guarantee optimal solutions. The net utility of BB-MDPOP approaches 97% of the best possible solution, which is achieved in a cooperative system when one can implement the optimal decision without charging agents. In comparison, R-M-DPOP is able to achieve around 89% of the best possible solution. Both algorithms improve significantly on the net utility achieved by the classic VCG mechanism, which is at 71% from the optimum.

The rest of this chapter is organized as follows: after preliminaries (Section 12.1), in Section 12.2 we move on to the issue of redistributing VCG payments. Section 12.2.1 introduces the R-M-DPOP algorithm, and Section 12.2.2 introduces the BB-M-DPOP algorithm. We present experimental results in Section 12.3, and then conclude.

12.1 Related Work

There is a long tradition of leveraging the VCG mechanism (or the *Clarke tax*) within Distributed AI, going back to Ephrati and Rosenschein[64–66], who introduced the use of the VCG mechanism into AI, and considered its role as a method to achieve consensus in multi-agent planning. For more recent work in Distributed AI that relates to the VCG mechanism, and more broadly the themes of mechanism design, we refer the reader to these surveys[103, 134], or to[41, 43, 116, 149, 150, 188, 220, 229].

The problem of tax waste in the context of the VCG mechanism was recognized early on in [64, 92, 209]. Well-known impossibility results[89, 101, 214] show that in general settings, for quasi-linear utility functions there can be *no* truthful mechanism that is efficient and exactly budget-balanced for all inputs.[7] Therefore, it has been assumed that VCG payments are impossible to redistribute back to the agents[208]. Nevertheless, some authors have suggested partitioning the population of agents into *independent groups*, which could pay VCG taxes to each other, such that overall budget balance is achieved[64, 88, 208]. Alternatively, in restricted settings, budget balance and efficiency were shown

[7] The so-called *d'AGVA mechanism*[7] offers exact budget balance *and* optimality. However, incentive compatibility is just a Bayes-Nash equilibrium, individual-rationality is obtained just *in expectation*, and the mechanism designer and the agents have to have common knowledge about a distribution on agent types.

possible[93, 115].

In our setting, the VCG mechanism always runs at a surplus, with the bank receiving a net payment from agents. A naive redistribution of these tax payments back to the agents would satisfy budget-balance but fail incentive compatibility. For instance, an agent might try to increase the tax payments made by other agents by overstating the negative impact of those agents on its own local solution in order to increase the payments they make and, in turn, its own share of these payments.

In some cases, together with the increase in problem size, it has been noted that the VCG payments tend towards zero[87, 130, 178]. Interestingly, we find in our experiments (and see also Faltings[68]) that this seems not to be the case, especially in structured domains.

To our knowledge, the first *truthful redistribution scheme* of VCG payments back to interested agents has been proposed by Bailey[9]. In Bailey's scheme, each agent receives payment T_{N-i}/N from the center, where T_{N-i} is the payment that the center would collect without agent i and there are N agents altogether. This is truthful because the redistribution is agent independent. Bailey studies the effect of this redistribution on the convergence of total payments towards zero, as an economy gets large through replication, demonstrating O($1/N^2$) error compared to O($1/N$) error without redistribution. The main limitations of Bailey's scheme are that it can sometimes run at a budget deficit, and also that it is a "macro-approach" rather than "micro-approach." Whereas Bailey considers only the use of signal T_{N-i} in determining the redistribution to agents, we determine redistributions based on detailed microstructure of a particular instance.

The basic idea of Bailey's scheme was rediscovered, and extended in different ways, by Porter et al.[174], Cavallo[29], and Guo and Conitzer[94]. Cavallo overcomes the shortcoming of Bailey's scheme (i.e. its potential budget deficit) in general settings by deriving a tight bound on what can be redistributed to an agent but again keeping a macro-view of the redistribution problem. Cavallo also formalizes explicitly (c.f. Porter et al.[174] in more restricted setting) the opportunity for redistribution of payments without compromising truthfulness. For his general results, Cavallo[29] imposes an anonymity requirement, which one can think of as providing a form of fairness: if two agents have the same potential for receiving a redistribution[8] then they should receive the same redistribution payment. We do not seek to achieve this fairness property in our scheme.

Porter et al.[174] study the related problem of *fair imposition*, in which costly tasks are to be allocated to a population of agents, costs are private information to agents, and the center will make transfers to the agents to provide incentive compatibility. They restrict attention to simple, non-combinatorial problems. Because of this setting of "imposition", the authors adopt the goal of fairness, trying to minimize the maximium loss of utility across all participants. Although a different problem to the one that we study, in order to achieve this they study what are, in effect, redistribution schemes for VCG mechanisms. Indeed, they briefly consider an alternate interpretation to a single-item auction

[8] Cavallo refers to this as the "surplus guarantee," it is a minimal bound on total payments made by other agents across all possible reports of agent i.

setting, where they rediscover the scheme of Bailey[9].

Guo and Conitzer[94] extend Cavallo's method[29], proposing a family of mechanisms with significantly better redistribution of payments. However, these mechanisms work only for a restricted setting: allocation problems with multiple, indistinguishable units, and agents with unit demand.

The other possibility for achieving budget balance [9] (following from[88]) is to *impose constraints* on the problem and then leverage these constraints to enable cross-payments across different parts of the system[64, 68]. By doing so, in principle one cannot guarantee efficiency anymore, but by carefully designing the constraints, budget-balance is possible. Ephrati and Rosenschein[64], study the problem of N agents trying to reach a consensus on a plan to transform the world from some initial state, to some final state for which they each have individual value. They propose to partition the group of agents in different sets, with each set forming a coordinated plan for some part of the larger problem and with payments flowing between different sets of agents. While similar in spirit to our approach, these authors do not provide details on how the partition can be formed by the agents in the first place, without providing an opportunity for manipulation. Moreover, whereas BB-M-DPOP imposes agent-wise, heterogeneous constraints on the problem, these authors seek to impose global constraints.

Faltings[68] studies the approach of Green and Laffont[88] by simply picking a random subset of agents (typically one), and excluding these from the decision but allowing them to receive payments. Reporting the first experimental results on structured problems, Faltings observe that while the total tax payments *increase* with the size of the problem, the cost of degradation due to removing one agent reduces, and finds a very significant net utility gain through this approach. Obviously, a drawback of this approach is that in a large optimization problem, some agent would not be considered at all in the entire problem. BB-M-DPOP extends this idea and is less draconian and more graceful. Each agent is able to receive *some portion* of the tax in return for *some* reduction of its influence on the solution. Rather than introduce a single, strong constraint for one agent we introduce individualized, weaker constraints for every agent.

12.1.1 The VCG Mechanism Applied to Social Choice Problems

Recall from Chapter 11 that the payment by agent i in the VCG mechanism is:

$$Tax(A_i) = \sum_{j \neq i} \left(R_j(X^*_{-i}) - R_j(X^*) \right) \tag{12.1}$$

$$= \sum_{j \neq i} Tax_j(A_i) = \sum_{j \neq i} R_j(X^*_{-i}) - R_j(X^*). \tag{12.2}$$

The disaggregation implied in this definition (with $Tax_j(A_i)$, to represent the payment made by agent A_i as a result of its marginal (negative) effect on the utility of agent A_j) is the same as the one used in

[9] Instead of seeking exact efficiency and trying to redistribute payments as best possible

M-DPOP. Each agent A_j can be relied upon to report each component of the total payment made by agent $A_i \neq A_j$. We recall from Section 11.4.1 that all instances of the social choice problem satisfy the property of *no positive externalities*:

$$\sum_{j \neq i} R_j(X^*_{-i}) \geq \sum_{j \neq i} R_j(X^*), \quad \forall A_i \in \mathcal{A} \tag{12.3}$$

An agent can only have the effect of changing the values of variables away from the best possible settings in the problem without the agent. This ensures that $Tax(A_i) \geq 0$ for all A_i, by Equation (12.1), so that the VCG payments collected by the center are always non-negative. One can conclude that the VCG mechanism in this setting of social choice always runs at a surplus.[10]

12.2 Incentive Compatible VCG Payment Redistribution

In this section we turn to the main goal of this chapter, which is that of payment redistribution. Specifically, our motivation is to reduce the loss of efficiency that is caused by having the bank receive net payments from the agents. As discussed in the introduction, in the standard VCG mechanism these payments must be wasted.

In the following, we consider two alternatives for dealing with the issue of VCG surplus. Both methods use the structure of the problem in determining the redistribution of payments. The first method (R-M-DPOP, Section 12.2.1) is in the spirit of Cavallo[29] and preserves the optimality of the solution, but is not guaranteed to achieve exact budget-balance: we redistribute only the components of the payments for which we can find a recipient that cannot possibly influence the particular component of the payment under consideration. The structure of the specific instance of the social choice problem is used to determine this possible influence.

The second method (Section 12.2.2) is in the spirit of Faltings[68] and does the inverse: it trades optimality for budget-balance. This method ensures that each payment is redistributable to some agent by deliberately breaking its influence on some part of the social choice problem. This in turn may affect optimality. Problem structure is used to determine where to break the influence of individual agents such that it is known which payments they can receive and also to best avoid compromising solution optimality.

[10] For comparison, notice that if the presence of some agent was to *increase* the range of values that can be assigned to some variable then it can have a positive externality on the rest of the agents. The VCG mechanism can run at a budget deficit in this kind of environment.

12.2.1 R-M-DPOP: Retaining Optimality While Seeking to Return VCG Payments

This first method preserves the optimality of the solution, but sacrifices budget-balance: we redistribute only the payments made by agents A_i for which we can find a recipient $A_l \neq A_i$ that cannot influence the marginal impact of A_i on the rest of the problem. Naturally, one necessary condition is that agent A_l's local solution is not itself influenced by the presence of agent A_i.

Here is a quick overview of this method:

1. As in M-DPOP, solve the main and marginal economies.

2. The optimal solution is implemented, just as in M-DPOP and VCG taxes are computed by each agent and reported to the bank.

3. In a new "redistribution phase" we seek to redistribute the tax payments back to the agents. For each agent A_i, we check for the specific problem instance whether some candidate recipient $A_l \neq A_i$ could have possibly influenced the computation of the tax payment. If not then it is safe to give the payment to A_l otherwise the payment accrues to the bank as in M-DPOP.

Let us consider a further disaggregation of the tax payments from Equation 12.2. Specifically, let us consider agent A_i and a single relation $r_j^k \in R_j$ that belongs to another agent A_j. Agent A_i will have to pay the following VCG tax for interfering just with A_j's relation:

$$tax_{r_j^k}(A_i) = r_j^k(X^*_{-i}) - r_j^k(X^*) \tag{12.4}$$

We call $tax_{r_j^k}(A_i)$ a *micropayment*. Summing up all micropayments over all agents $A_j \neq A_i$ gives the VCG tax payment made by A_i:

$$Tax(A_i) = \sum_{j \neq i} \sum_{r_j^k \in R_j} tax_{r_j^k}(A_i) \tag{12.5}$$

We abuse notation in the following, and write $r_j^k \in Tax_j(A_i)$ if the payment $tax_{r_j^k}(A_i) \neq 0$ and adopt $scope(Tax(A_i))$ to denote the set of variables involved in relations r_j^k for some agent $A_j \neq A_i$, i.e. $r_j^k \in Tax_j(A_i)$. This is the set of variables whose values are influenced by the presence of agent A_i in a way that changes the utility of some other agent, and thus impacts the payment by agent A_i.

Designate an agent $A_l \neq A_i$ as a *candidate* to receive as a refund the entire VCG payment $Tax(A_i)$ made by agent A_i. This candidate agent needs to be chosen independently of the declarations of any agent that can *possibly* affect $Tax(A_i)$. Our algorithm does this as follows (see Algorithm 28):

1. **Restrict areas of direct influence**: for each agent A_r, restrict to $P(A_r)$ the variables on which the agent can express interest and ignore its declarations when they involve other variables.

2. **Select a candidate to receive a refund**: for each A_i, designate an agent A_l as a candidate to receive the payment $Tax(A_i)$ made by A_i, by random selection[11] among agents that cannot possibly have a direct influence on any variable involved in $P(A_i)$ (i.e. $P(A_l) \cap P(A_i) = \emptyset$).

3. **Check possible indirect influence**: given candidate A_l for $Tax(A_i)$ check A_l has no possible (indirect) influence on the values of any variables in $scope(Tax(A_i))$ in either $DCOP(\mathcal{A})$ or in $DCOP(-A_i)$. If there is no possible influence, A_l receives the payment $Tax(A_i)$ as a refund and otherwise the payment accrues to the bank.

Step 1, which restricts the impact of an agent's messages, is without loss given that $P(A_i)$ is the set of variables on which an agent A_i can possibly have interest. Step 2 is a specific example of a more general idea: we must pick the candidate A_l by some criterion that is not related to agent declarations. The algorithm for performing the check on indirect influence is presented in Section 12.2.1.2.[12]

This mechanism cannot guarantee budget-balance since it can happen that A_l can have a possible influence on some of the variables in $scope(Tax(A_i))$ and therefore the tax payment made by agent A_i. However this approach can significantly reduce the payments that need to be made, as we see in the experimental results in Section 12.3.

Theorem 11 *The R-M-DPOP algorithm is a faithful distributed implementation of efficient social choice, never runs a budget deficit, remains individual-rational for agents and can redistribute some of the VCG payments collected by the bank back to the agents.*

PROOF. Faithfulness follows from the faithfulness of M-DPOP and because agent A_l cannot influence whether or not it receives as a refund the tax payment made by some other agent. This is by construction. To see that the mechanism never runs a budget deficit note that each agent's tentative tax payment to the center remains non-negative by Eq. (12.3), but that sometimes the center simply returns this payment to some other agent. For individual-rationality (IR), recall that the VCG is IR because the payment Eq. (12.1) is less than an agent's utility for solution X^*. The difference here is that agents sometimes receive an additional payment, when they are eligible to receive the tax payment of some other agent. □

[11] For instance, this random selection can be done using a secure distributed protocol for random number generation, like Benaloh[17]: each $A_j \neq A_l$ proposes a random number r_j between 0 and $|\mathcal{X}|$. All numbers are added up and the result is the sum modulo $|\mathcal{X}|$: $rnd = \sum_{A_j} r_j \mod |\mathcal{X}|$. rnd is then the ID of the chosen agent.

[12] The mechanism only makes a single attempt to find an agent that is eligible to receive the tax payment by agent A_i. If the candidate agent chosen does not qualify the tax is not redistributed and goes to the bank. To increase the chances for redistribution one might think to select a *group* of candidate agents, with each agent then checked for eligibility. Successful agents could then split the tax among themselves or get the entire tax with some probability. However, each candidate chosen in Step 2 would have an interest to make the *other* candidate agents have a possible influence in order to increase its own chance of receiving a refund. This would significantly complicate the procedure.

Algorithm 28 *R-M-DPOP with VCG refunds: towards budget-balance*

Inputs: Bank knows community membership $P(A_i) \forall A_i \in \mathcal{A}$
Outputs: Each VCG tax is refunded to an agent, or wasted

run M-DPOP algorithm to compute solution and VCG payments

Procedure TAX_refunds

forall A_i **do**

  Bank selects an agent A_l randomly s.t. $P(A_l) \cap P(A_i) = \emptyset$
  Agents execute $LABEL^l(DCOP(\mathcal{A}))$ on $DCOP(\mathcal{A})$ (see Section 12.2.1.2)
  if *possible influence of* A_l *on* $\forall X_k \in scope(Tax(A_i))$ **then** waste $Tax(A_i)$
  else

    Agents execute $LABEL^l(DCOP(-A_i))$ on $DCOP(-A_i)$ (see Section 12.2.1.2)
    if *possible influence of* A_l *on* $\forall X_k \in scope(Tax(A_i))$ **then** waste $Tax(A_i)$
    else refund $Tax(A_i)$ to A_l

12.2.1.1 An example of possible, indirect influence

Consider the example from Figure 12.2. The figure illustrates a DFS arrangement of a DCOP problem (the largest triangle in the figure) and fixes an agent A_l and the tax payment made by some agent A_i. Agent A_l is restricted to placing relations only on the subset of variables $P(A_l)$ for which it has possible influence. Let H_l denote the lowest node in $DFS(\mathcal{A})$ such that its subtree, $T(A_l)$, contains *all* the nodes on which A_l is allowed to place relations.[13] It follows that A_l can have no *direct* influence on nodes outside of $T(A_l)$, including any sibling or ancestor of H_l.

The question addressed in checking for *possible* influence is the following: *for which variables outside of* $T(H_l)$ *was it possible for agent* A_l *to have an indirect influence on the values assigned?* To make this example concrete we assume that variable H_l can take three values `a, b, c`. Let us assume that A_l can completely control H_l through its relations placed in the subtree $T(A_l)$ (this is the worst case scenario). Let Y be the ancestor of H_l in the DFS ordering with possible values `d, e, f`, and assume that some other agent has imposed a relation between H_l and Y, as depicted in Table 12.1.

Assuming omnidirectional UTIL propagation as explained earlier in Section 4.1.6, in addition to a *UTIL* message from T_l, node Y will also receive *UTIL* messages from all its other subtrees and also from its parent, Z. Let us assume that the sum of all these *UTIL* messages other than from H_l arriving at Y is the vector `(5,5,5)`, giving the utilities for values of $Y \in$`(d, e, f)` in the rest of the problem. Notice that this vector cannot be influenced by A_l, since A_l could not place any relations outside $T(A_l)$.

[13] Notice that the subtree $T(A_l)$ does not necessarily include all siblings of the local variables of A_l. Therefore, there could be variables above H_l which are connected with equality constraints with variables below H_l, and thus under the influence of A_l. However, the scheme we propose in Section 12.2.1.2 would detect such influence.

$Y =$	d	e	f
$H_l = a$	3	2	1
$H_l = b$	2	3	1
$H_l = c$	4	3	2

Table 12.1: *Example of possible influence. Here, node H_l is the node that defines the subtree isolating the direct influence by A_l and Y is the ancestor of H_l in the DFS ordering. $r(H_l, Y)$ is the relation between H_l and Y, and owned by an agent other than A_l.*

Y=	d	e	f
$H_l = a$	$\mathbf{8 + U_l(H_l = a)}$	$7 + U_l(H_l = a)$	$6 + U_l(H_l = a)$
$H_l = b$	$7 + U_l(H_l = b)$	$\mathbf{8 + U_l(H_l = b)}$	$6 + U_l(H_l = b)$
$H_l = c$	$\mathbf{9 + U_l(H_l = c)}$	$8 + U_l(H_l = c)$	$7 + U_l(H_l = c)$

Table 12.2: *$JOIN^l_{Y \rightarrow Z}$: table with global utilities for combinations of assignments $\langle H_l, Y \rangle$. A_l can claim any utility on each value of H_l, and in turn can influence whether $Y = d$ or $Y = e$ is optimal. The assignments that can be selected by Y are represented as the red cells.*

However, it remains possible that agent A_l can indirectly influence the value selected for Y through the utilities it assigns to the three different values of H_l and through its impact on the choice on value of H_l. Letting these utilities be $U_l(H_l)$, and factoring the utilities reported in the rest of the problem, the propagation would choose the maximum in each row of Table 12.2, as indicated in bold.

The chosen column (the value for Y) depends on the utilities A_l assigns to $U_l(H_l)$. Notice that A_l can never force $Y =$ `f`, since this will never give the maximum utility. However, A_l can still influence Y to take value $Y = d$ by assigning a large utility to either $H_l = a$ or $H_l = c$, and force $Y = e$ by assigning a large utility to $H_l = b$. Thus, there is possible influence and any tax collected because of some other agent's influence on Y cannot be given to A_l without breaking the incentive properties. It is not possible to prove, in this case, that the agent has a lac of influence. Had this been possible, then any tax collected from other agents for their influence on Y could have been given to A_l.

Agent A_l's indirect influence on a variable outside $P(A_l)$ depends on the preference of the other agents. For instance, if the utilities from the rest of the problem for the values of Y would instead aggregate to the vector `(5,5,1000)` then the influence of A_l over H_l is not enough to prevent Y from taking value f; therefore, A_l would have no influence whatsoever on Y.

12.2.1.2 Detecting Areas of Indirect, Possible Influence

We present in this section an algorithm (Algorithm 29) to check whether an agent A_l can possibly influence the value taken by a variable X outside of its area of direct influence. This algorithm is used as a subroutine in R-M-DPOP (lines 4, 5 and 7, 8) in checking whether a candidate agent is actually eligible to receive a payment redistribution. By this we mean that we *can prove*, given the reported preferences of other agents, that the same value of X would be selected by M-DPOP for all messages that could have been sent by agent A_l, and therefore A_l has no influence on X.

First we introduce the following *LABEL* data structure to keep track of the possible (indirect) influence an agent A_l can have on variables in the rest of the problem:

Definition 46 (Influence label) *We can characterize the influence that an agent A_l has on a group of variables by an **influence label**, which is a multidimensional matrix with one dimension for each variable in the group. Each element of the label corresponds to a combination of values for the variables (a tuple), and takes value 1 if A_l can force the corresponding tuple to be chosen in the optimal solution, and 0 otherwise.*

Notice that in the optimal solution, any variable X_k will take some value, so all labels will contain at least a "1" for that combination. Refer to Figure 12.1. From the most influence to the least, a label can have:

1. 1's for all elements: this means that A_l can fully influence all variables in the label, and can impose any value combination - e.g. Figure 12.1(1).
2. 1's for at least 2 different values for each variable: this means that A_l can (partially) influence all variables in the label - e.g. Figure 12.1(2).
3. 1's for just a single value v of a certain variable X_k: this means that A_l has no influence on X_k: no matter what A_l does, X_k always takes value v ($X_b = f$ is in this situation in Figure 12.1(3)).
4. A single 1: then A_l cannot influence any of the variables at all; they will take the same values regardless of what A_l does- e.g. Figure 12.1(4).

It is easy to see that when A_l can have a direct relation on a variable then it can claim any utilities it desires for all values and make any value the best one for the system. Therefore the labels are initialized to "1" on each of the values for such a variable.

We now describe the *label propagation* process that computes and propagates *LABEL* messages to determine where the influence of an agent stops. The *LABEL* propagation determines the possible influence of A_l on all variables in the problem. It is run as a post-processing step once M-DPOP has completed to determine which candidate agents are eligible to receive tax re-distributions. This was

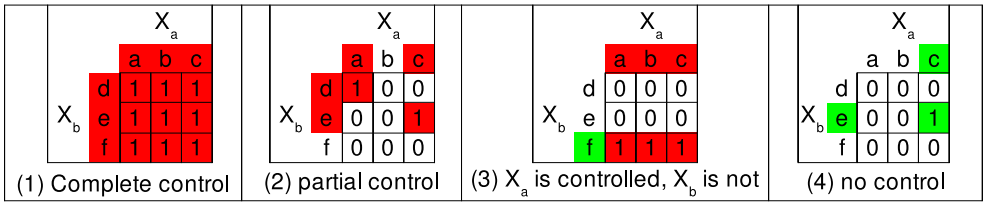


Figure 12.1: *Detecting Areas of Indirect Influence: Example labels for a set of 2 variables X_a and X_b. (1) A_l can impose any combination of values for X_a and X_b. (2) A_l can impose just 2 value combinations: either $\langle X_a = a, X_b = d\rangle$, or $\langle X_a = c, X_b = e\rangle$. (3) A_l can impose any value for X_a, but X_b takes value f regardless. (4) A_l can do nothing: $\langle X_a = c, X_b = e\rangle$ is chosen.*

Algorithm 29 *Computing and sending **LABEL**s: determining A_l's influence.*

Procedure LABEL_passing for A_l

1 Node X_t gets $LABEL^l_{j\to t}$ from neighbor X_j
2 **forall** $X_m \in \{P_t \cup C_t \setminus X_j\}$ **do**
3 $JOIN^l_{t\to m} = r^t_j \oplus \left(\bigoplus_{X_q \in \{P_t \cup C_t \setminus X_j, X_m\}} UTIL_{q\to t}\right)$ (detailed in text in 2.a)
4 $LABEL^l_{t\to m} = JOIN_{t\to m} \perp LABEL^l_{j\to t}$ (*this keeps in $LABEL^l_{t\to m}$ only the dimensions from $UTIL_{m\to t}$ and is detailed in text in 2.b*)
5 send $LABEL^l_{t\to m}$ to X_m

used in Algorithm 28. In overview, the *LABEL* propagation starts from the nodes on the border of the $P(A_l)$ area: the node H_l, which is the lowest node whose subtree includes $P(A_l)$, and the nodes L^j_l which are the highest nodes in $T(A_l)$ which do not contain any variables of A_l in their subtree. The propagation proceeds outwards as far as the agent A_l still has a possible influence. Payments originating as the result of values on variables outside A_l's area of possible influence can be safely refunded to A_l without breaking the incentive properties.

Consider a node X_t that has just received a LABEL message from one of its neighbors, X_j. The LABEL message summarizes the possible influence A_l could have on X_j. This information, together with the aggregation of the other agents' utilities for values of X_t, can determine the possible influence A_l could have on X_t, i.e. its LABEL, which will be sent further on to its neighbors, and so on. To this end, we use the *omnidirectional* utility propagation introduced in Section 4.1.6.

Denoting as $LABEL^l$, the propagation that occurs for A_l, it works as follows:

1. H_l and all $L^1_l, \ldots, L^k_l$ are determined to delineate the area of direct influence of A_l: it is easy for these nodes to identify themselves as a direct result of the DFS construction phase. Notice that they are chosen such that they are not owned by A_l.

2. H_l and all $L^1_l, \ldots, L^k_l$ initialize the $LABEL^l$ propagation by constructing *LABEL* messages filled with 1's and sending them to their tree-neighbors outside the $P(A_l)$ area;

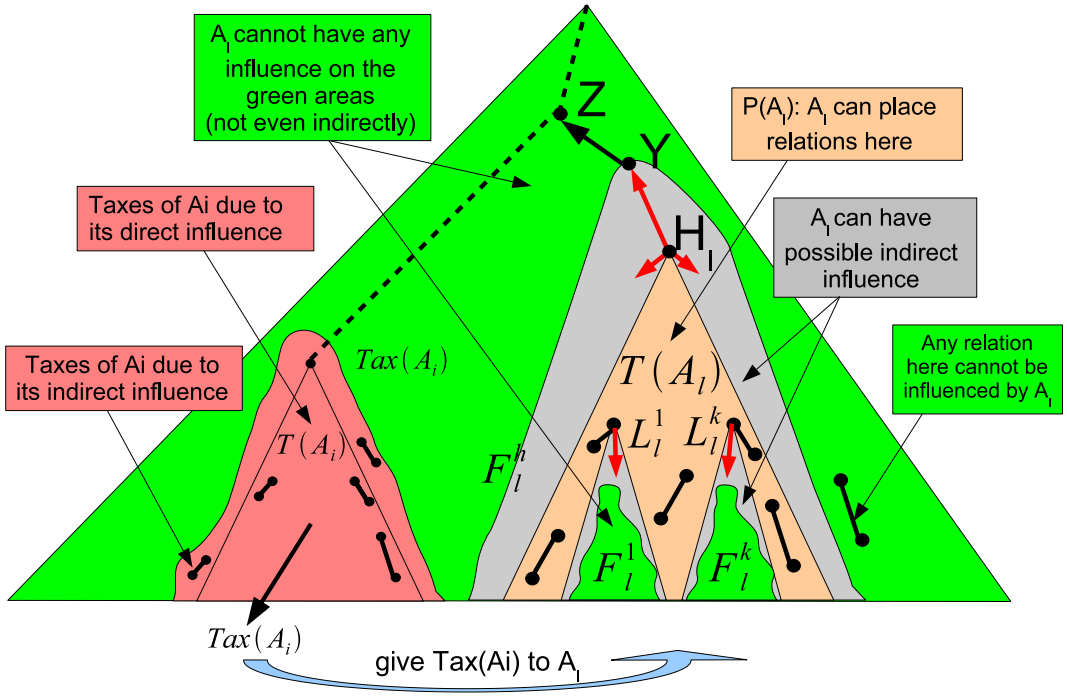


Figure 12.2: *Checking possible influence: A structural method to determine whether it is safe to redistribute the VCG payment of an agent A_i to another agent A_l. Agent A_l must be unable to influence any component $tax_{r_j^k}(A_i)$ of $Tax(A_i)$ for $A_j \neq A_i$ and $r_j^k \in R_j$. Let $P(A_l)$ denote the subset (orange area in the figure) of variables on which A_l is allowed to place direct relations. A_l can also indirectly influence other variables, via other agents' relations (the gray areas in the figure). However, its indirect influence is limited, and the green areas are completely out of A_l's influence. Since all components of $Tax(A_i)$ (contained in the red area) are outside A_l's influence, it is safe to give A_l the VCG tax of A_i.*

3. Subsequently, all nodes wait for incoming $LABEL^l$ messages, compute their own labels for their tree-neighbors not in the $P(A_l)$ area, and propagate these to their neighbors.

For Step 3 a node X_t performs Algorithm 29 to compute and propagate its own labels. In overview, this algorithm works as follows:

1. Node X_t receives a message $LABEL^l_{j\to t}$ from X_j
2. Node X_t computes $LABEL^l_{t\to m}$ for each one of its tree neighbors $X_m \neq X_j$:
   (a) It joins the following: (1) the relation r_j^t it has with the sender of the LABEL message, (2) all the UTIL messages it has received from its tree neighbors except the one from the sender of the LABEL message (this UTIL message is presumably manipulated by A_l) and (3) the UTIL message sent by X_m. The result is $JOIN^l_{t\to m}$.
   (b) For all tuples marked as 1 in $LABEL^l_{j\to t}$, perform the corresponding slice operation in $JOIN^l_{t\to m}$. The resulting hypercubes are then projected onto X_t. These projections are

the optimal results that will be selected in each of the possible deviations by A_l. Therefore, they are marked with 1's in the corresponding outgoing $LABEL^l_{t \to m}$ message.

(c) Node X_t sends $LABEL^l_{t \to m}$ to its tree neighbor X_m

Let us now refer again to the example from Figure 12.2. Since we assumed that A_l can impose any value on H_l, A_l's label for H_l is thus `(1,1,1)`. By design, H_l is not controlled by A_l and is expected to propagate this label *correctly* to its parent, Y, which receives it in Step 1 of Algorithm 29. Y then performs Step 3 of Algorithm 29: it joins all the UTIL messages received from all its other tree neighbors except H_l (i.e. its parent Z, and its children other than H_l). Assume as before that this produces the vector $(5, 5, 5)$. Y also adds the relation it has with H_l to this join. The result is $JOIN^l_{Y \to Z}$, depicted in Table 12.2. Y then computes its $LABEL^l_{Y \to Z}$ message for its parent Z as in Steps 4 and 5 of Algorithm 29. Each tuple from $LABEL^l_{H_l \to Y}$ that is associated with a "1" is considered in turn (actually, this is all of them: $H_l = a$,$H_l = b$,$H_l = c$). For each one, we perform a slice in $JOIN^l_{Y \to Z}$: this results in the corresponding rows of Table 12.2. For each row, we project on to Y, i.e. we find the best assignment for Y. This assignment of Y is enforceable by A_l, and thus is assigned a "1" in $LABEL^l_{Y \to Z}$. Concretely, $H_l = a$ forces $Y = d$, $H_l = b$ forces $Y = e$ and $H_l = c$ forces $Y = d$, thus $LABEL^l_{Y \to Z} = (1, 1, 0)$.

Note: had A_l's label for H_l been `(1,0,1)`, its label for Y would have been `(1,0,0)`, meaning that only $Y =$ `d` is possible and thus A_l would have had no possibility to influence Y's value.

Finally, Y sends this label to its parent, Z, and the process continues until the labels contain just a single value of 1. Note that the number of "1"s in a label can never increase during such a propagation, since for every choice of input value there can be only one optimal output value. This means that the propagation will eventually converge to labels with a single "1". By propagating labels in the same way as propagating messages in M-DPOP, we can determine the set of variables that an agent can potentially influence.

Lemma 3 *$LABEL^l$ propagation is non-manipulable by A_l, and conservatively determines A_l's influence on all variables in the SCP.*

PROOF. A_l is not involved in any computation or message passing during the $LABEL^l$ propagation. The propagation is initiated by the nodes H_l and L^*_k, which are *conservatively* chosen, outside the area of direct influence of A_l. They are not under A_l's control, thus expected to initiate the propagation with *LABEL* messages containing all 1's (i.e. assuming the worst case, when A_l can impose any value on them). The propagation then proceeds *outside* the area of influence of A_l, through nodes which A_l does not control, and therefore, expected to propagate *LABEL* messages correctly. □

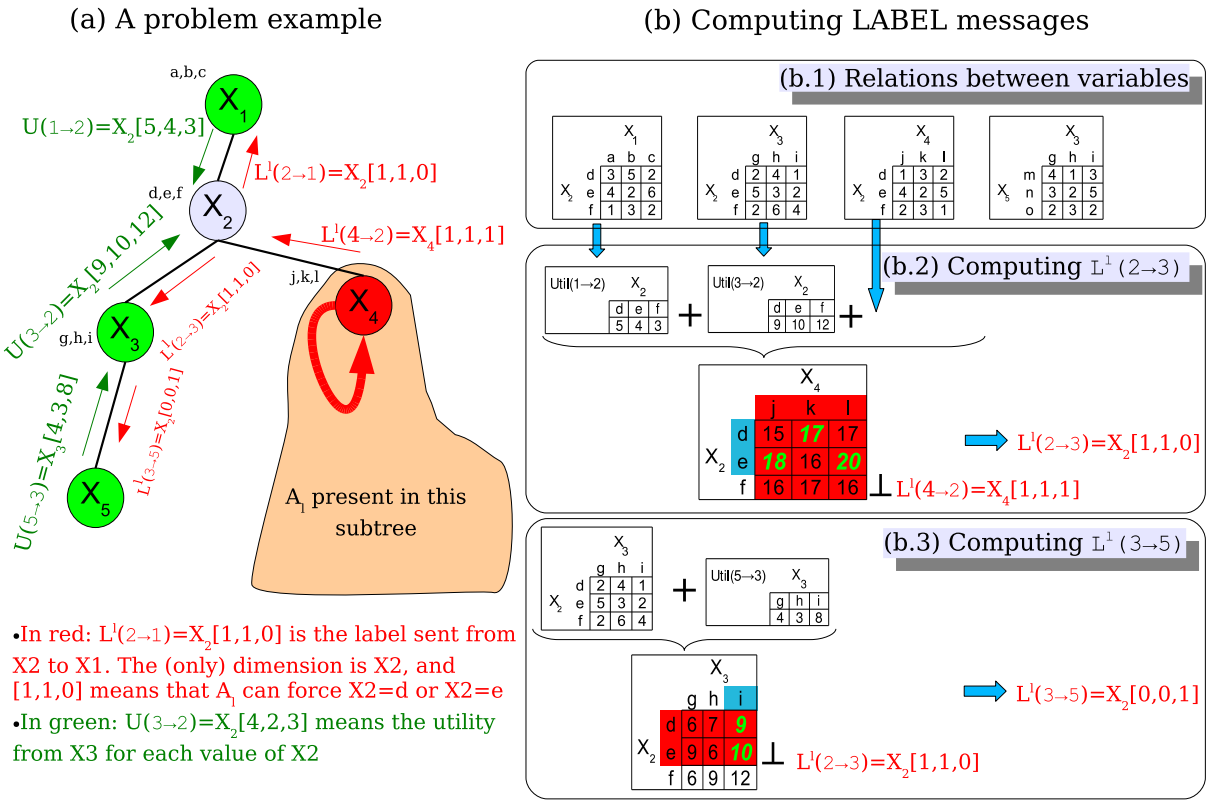


Figure 12.3: *A concrete numerical example of LABEL propagation. A_l is present in the subtree rooted at X_4 and has direct influence on X_4. Thefore the label* `[1,1,1]` *is passed to X_2. X_2 computes its label by joining the UTIL messages it gets from its neighbors other than X_4, and considering the relation r_4^2 X_2 shares with X_4 and $LABEL_{4\rightarrow 2}$. X_3's label is* `[0,0,1]`*, meaning that A_l cannot influence at or below X_3 and taxes originating at or below X_3 can be redistributed to A_l.*

12.2.1.3 A concrete numerical example of LABEL propagation

We show in Figure 12.3 a concrete example on which to illustrate the use of *LABEL* propagation. We determine the possible influence for A_l.

As seen in the figure, A_l's presence in the problem is limited to the subtree rooted at X_4. Therefore, the worst-case scenario is assumed: A_l can completely influence X_4 to take any value it desires. The $LABEL^l$ message that X_4 generates and sends to X_2 is thus `[1,1,1]`.

X_2 computes the join of the UTIL messages it receives from X_1 and X_3, and of the relation it shares with X_4. This computation is shown in Figure 12.3(b)-middle. The result is a matrix with 2 dimensions, X_2 and X_4, which states what values X_2 will take as a function of the values X_4 takes. This can be influenced by A_l according to the $LABEL^l_{4\rightarrow 2}$, i.e. A_l can force any column in that matrix. However, for both $X_4 = j$ and $X_4 = l$ then X_2's optimal value is the same, $X_2 = e$. Notice that A_l has no way of forcing $X_2 = f$ via its influence on X_4. Therefore, the $LABEL^l$ message computed by X_2 is `[1,1,0]`. X_2 sends this message to its neighbors, X_1 and X_3.

Figure 12.3(b)-bottom shows a similar computation performed by X_3. X_3 joins the $UTIL_5^3$ message it received from X_5 and the relation it shares with X_2. The result is a matrix with 2 dimensions, X_2 and X_3, which states what values X_3 will take, as a function of the values X_2 takes. This can be influenced by A_l according to the $LABEL^l_{2\rightarrow 3}$, i.e. A_l can force either $X_2 = d$, or $X_2 = e$, but not $X_2 = f$. However, for both $X_2 = d$, and $X_2 = e$, X_3's optimal value is the same, i.e., $X_3 = i$. Notice that A_l has no way of forcing $X_3 = g$ or $X_3 = h$ via its influence on X_2 via X_4. Thus, the $LABEL^l$ message computed by X_3 is `[0,0,1]`, meaning that the value of X_3 cannot actually be influenced *at all* by A_l. Therefore A_l's influence stops alltogether at and below X_3. This means that whatever (micro)payments some agent $A_i \neq A_l$ has to pay for its effect on variables X_3 or below, these taxes can be safely redistributed to A_l, as A_l has no way of influencing them.

12.2.2 BB-M-DPOP: Exact Budget-Balance Without Optimality Guarantees

The redistribution method described so far guarantees the optimality of the final solution to the social choice problem. Because of this, it is unable to ensure complete budget-balance, and the protocol may run at a budget surplus to the bank and thus the population of agents may continue to lose utility through these payments. In this section we propose a scheme that guarantees complete budget balance, at the expense of the optimality.

Similarly to Faltings[68], we add constraints to the problem to prevent an agent A_l from influencing a part of the taxes by preventing it to have even possible influence on the part of the problem with variables in the scope of the tax payment. However, recall that we aim to allow A_l to have *some* influence in a restricted part of the problem, unlike Faltings, where A_l is excluded altogether. We achieve this by assigning *a priori* to each agent A_i another agent A_l from another part of the problem, who will collect A_i's payment. During the optimization, we need to put A_l and A_i in well separated parts of the problem; specifically, we must ensure (via constraints) that A_l cannot have any influence on the marginal impact that agent A_i has on some subset of the variables, and thus on the tax payments. We do this by performing two versions of the *UTIL* propagation: one with A_l's relations taken into account, and another one that does not take A_l's relations into account.[14] Intuitively, the propagations *with* A_l's relations considered are used in a "surrounding" area to A_l, to allow it to express its preferences on a subset of the problem. Beyond this "surrounding" area to A_l, the propagations *without* A_l's influence are used, thus effectively eliminating its influence.

Let us refer to the example from Figure 12.4. The example illustrates a problem arranged as a DFS tree, and two agents, A_i and A_l in separate parts of the problem. The mechanism decides *a priori* (i.e., before any messages are received) that the VCG tax of A_i will be given to A_l. Therefore, we need to ensure that A_l has no way of affecting the impact of A_i on the rest of the problem and thus on the tax payment made by A_i. We achieve this by including constraints, in the following way:

[14] The second propagation is similar to the marginal propagations executed in M-DPOP for the marginal economy without A_l.

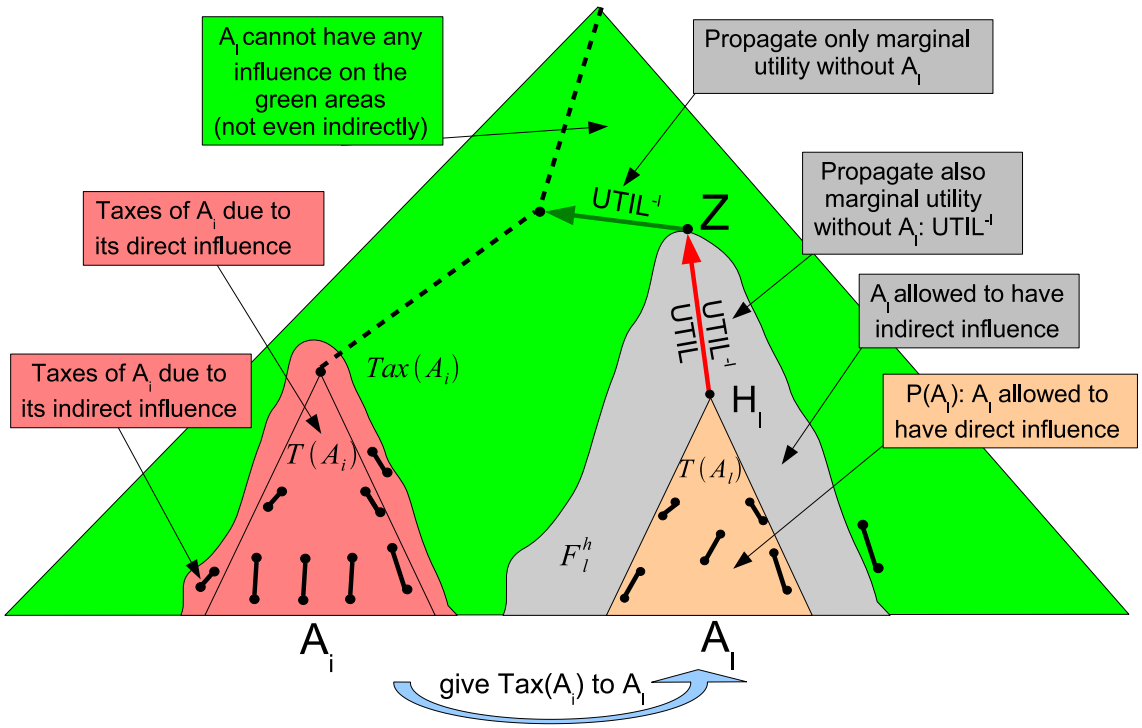


Figure 12.4: *Exact budget-balance in return for possible loss of optimality guarantees: a structure-based method to forcibly limit an agent A_l's influence in the problem. A_l is also allowed to indirectly influence other variables as well, via other agents' relations (the gray areas). Beyond a certain area, A_l has its influence forcibly eliminated. This area is defined by the cutoff point Z beyond which only marginal UTIL messages that do not contain any influence from A_l are propagated. These are the green areas and cannot be influenced by A_l. Since all components of $Tax(A_i)$ (contained in the red area) are outside A_l's possible influence it is safe to give A_l the VCG tax of A_i.*

1. We allow A_l to post its relations normally on variables in $P(A_l)$, and within $T(A_l)$ (the minimal subtree which contains $P(A_l)$) the normal *UTIL* and *VALUE* propagations take place.

2. From H_l (the root of $T(A_l)$) we propagate upwards two versions of the *UTIL* messages: the normal *UTIL* (optimal utilities, including the influence of A_l) messages, and $UTIL^{-l}$ messages (sent in solving the problem without A_l.)

3. A cutoff point, Z is chosen in any fashion that is independent of A_l's declarations. Influence of A_l is permitted in the subtree rooted at Z (which is the gray area in Figure 12.4). On the path from H_l to Z, we propagate both versions of the UTIL messages (with and without A_l's relations included). During the downward VALUE propagation, we select optimal values for the variables on the path from Z to H_l according to the UTIL messages that contain also A_l's influence. This ensures that we allow A_l to express its (indirect) preferences in the subtree rooted at Z.

4. Outside the Z-rooted subtree, A_l's influence is prevented by using just the marginal *UTIL* message $UTIL^{-l}$. This constraint has the effect that the values chosen for all variables outside the

Algorithm 30 *BB-M-DPOP: budget-balanced distributed mechanism for social choice*

Inputs: Bank knows community membership $P(A_i), \forall A_i \in \mathcal{A}$
Outputs: a (possibly suboptimal) solution; each VCG tax is refunded to an agent;

forall A_i **do**

1 Bank selects an agent A_l s.t. $P(A_i) \cap P(A_l) = \emptyset$
2 A_l will receive A_i's VCG payment, $Tax(A_i)$
3 Select cutoff point for A_l, let this be node Z
4 In subtree rooted at H_l, execute normal *UTIL/VALUE*
5 From H_l up to Z, propagate main and marginal UTIL ($UTIL, UTIL^{-l}$)
6 From Z down to H_l, propagate main VALUE (i.e. consider A_l's influence)
7 From Z onwards propagate just marginal $UTIL^{-l}/VALUE^{-l}$(exclude A_l)
8 Compute VCG (micro)payments normally
9 Any payments issued outside of the subtree rooted at Z can be given to A_l

subtree rooted at Z are independent of A_l. This makes it safe to give A_l the VCG tax of A_i , since A_l cannot influence its computation.[15]

12.3 Experimental evaluation

We present the results of an experimental evaluation of R-M-DPOP and BB-M-DPOP in a distributed meeting scheduling problem. The problems consist of agents working for a large organization and representing individuals, or groups of individuals, for the purpose of scheduling meetings for some upcoming period of time. Although the agents themselves are self interested, the organization as a whole requires an optimal overall schedule that minimizes cost (alternatively, maximizes the utility of the agents). This motivates the need for a faithful distributed implementation such as M-DPOP, rather than a cooperative approach such as vanilla DPOP. In enabling this, we can imagine that the organization distributes a virtual currency to each agent (perhaps using this to prioritize particular participants.)

Each agent A_i has a set of local *replicate* variables X_j^i for each meeting $M_j$in which it is involved. The domain of each variable X_j (and thus local replicas X_j^i) represents the feasible time slots for the commonly known meeting. An equality constraint is included between replica variables to ensure that meeting times are aligned across agents. If a meeting has q participants, it is sufficient to create $q-1$ equality constraints that connect the corresponding variables in a linear chain. Since an agent cannot participate in more than one meeting at once there is an *all-different* constraint on all variables X_i^j belonging to the same agent. This is modeled as a clique constraint between these meeting variables. Each agent assigns a utility to each possible time for each meeting by imposing a unary relation on

[15] A side effect is that A_l's own VCG taxes outside the tree rooted at Z effectively become 0 as A_l no longer has any influence on these variables.

each variable X_j^i. Each such relation is private to A_i, and denotes how much utility A_i associates with starting meeting M_j at each time $t' \in d_j$, where d_j is the domain for meeting M_j. The social objective is to find a schedule in which the total utility is maximized while satisfying the all-different constraints for each agent.[16]

Following Maheswaran et al.[127], we model the organization by providing a *hierarchical structure*. In a realistic organization, the majority of interactions are within departments, and only a small number are across departments and even then these interactions will typically take place between two departments adjacent in the hierarchy. This hierarchical organization provides structure to our test instances: with high probability (around 70%) we generate meetings within departments, and with a lower probability (around 30%) we generate meetings between agents belonging to parent-child departments. We generated random problems having this structure,with an increasing number of agents: from 5 to 50 agents.[17] Each agent participates in 1 to 5 meetings, and has a uniform random utility between 0 and 10 for each possible schedule for each meeting in which it participates. The problems are generated such that they have feasible solutions.

For each problem size, we averaged the results over 10 different instances. All experiments were performed in the FRODO multiagent simulation environment[154], on a 2.0Ghz/1GB RAM laptop. FRODO is a simulated multiagent system, where each agent executes asynchronously in its own thread, and communicates with its peers only via message exchange.

We experiment with both classes of redistribution schemes: R-M-DPOP which guarantees optimality but not budget balance and BB-M-DPOP, which guarantees budget-balance at the expense of optimality.

12.3.1 R-M-DPOP: Partial redistribution while maintaining optimality

This set of experiments analyzes the redistribution potential of the R-M-DPOP scheme. The results are presented in Figure 12.5. As the problems grow in size, we observe an increase in the percentage of taxes that can be redistributed by R-M-DPOP. The intuition is simple: as the problems grow in size, it is more likely that each agent's influence spans only a limited area in its neighborhood. Therefore, it is more likely to find a recipient from a different part of the problem for any VCG tax, such that the recipient has no influence on the tax. This is why the percentage of redistribution increases with the problem size.

Figure 12.6 compares the *net* efficiency of the optimal solution, the R-M-DPOP algorithm, and the VCG mechanism. In the VCG mechanism, each agent's net utility is the difference between the utility it derives from the solution which is being chosen (in this case the optimal one) and its VCG tax. In

[16] In a simple variation one could also seek to maximize the weighted utility across the agents, wherein some agents receive more priority within the organization than other agents. The VCG payments, and also M-DPOP, can be easily extended to provide appropriate incentives in this setting.

[17] Available at http://liawww.epfl.ch/People/apetcu/research/mdpop/MSexperiments.tgz

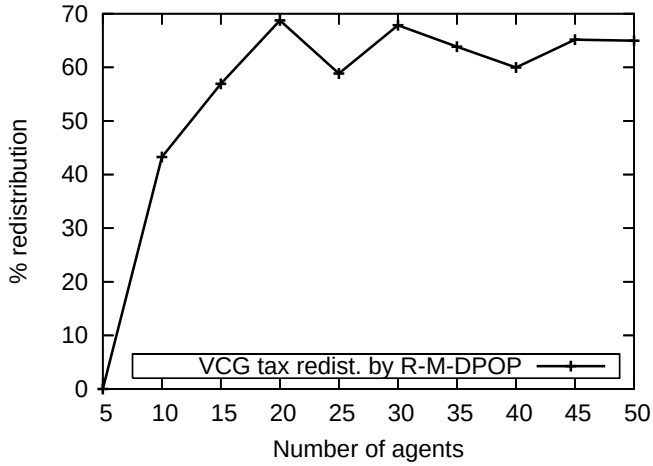


Figure 12.5: *The percentage of the amount of VCG taxes that can be redistributed by R-M-DPOP increases with the size of the problem.*

R-M-DPOP, some agents receive tax refunds from the bank, which get added to their net utility. We can see that the loss in utility while using the VCG mechanism (with respect to the optimal solution in a cooperative system) is quite significant, and increases with the size of the problem. This is because as the problems increase, competition increases and more payments are collected. In contrast, R-M-DPOP manages to redistribute a significant percentage of these payments back to the agents, thus limiting the net utility loss.

12.3.2 BB-M-DPOP: Complete redistribution in exchange for loss of optimality

This set of experiments analyzes the tradeoff introduced by BB-M-DPOP between the loss of optimality of the solution and the utility gain induced by the reimbursement of the VCG taxes. The results are presented in Figure 12.6. We notice that BB-M-DPOP fares much better than the VCG mechanism, and the overall utility when using BB-M-DPOP is very close to the optimal utility without any taxes. This is in spite of the fact that BB-M-DPOP does not guarantee optimal solutions. It compensates for this by returning *all* VCG taxes back to the agents, thus avoiding the net utility loss incurred by burning these taxes.

As the problems grow in size, it is more likely that each agent's influence spans only a limited area in its neighborhood. Therefore, it is more likely to find a recipient from a different part of the problem for any VCG tax, such that the recipient has limited or no influence on the tax anyway, and therefore cutting off its influence does little to change the optimal solution.

Interestingly, BB-M-DPOP outperforms R-M-DPOP in terms of the net utility to agents. This shows that in this environment it is more beneficial to accept a small loss in optimality and be able to redistribute all VCG taxes than insisting on optimality and thereby forfeiting the guarantee of budget-

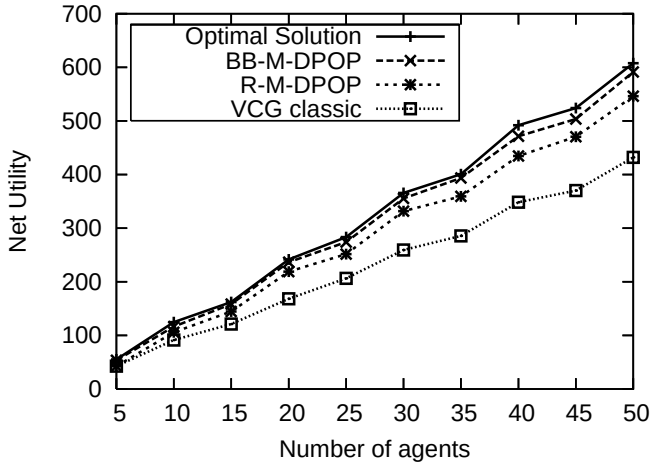


Figure 12.6: *The overall net utility of the agents in the system. The classical VCG mechanism incurs losses in overall net utility compared to the optimal solution as the agents have to pay the taxes to the bank. R-M-DPOP reduces the losses by redistributing some of the VCG payments back to the agents. BB-M-DPOP offers even better overall utility, as the loss of optimality is counterbalanced by the complete redistribution of the VCG payments.*

balance.

Figure 12.7 shows the amount of computational effort required by R-M-DPOP and BB-M-DPOP compared to M-DPOP. Here, computational effort means the total size of the UTIL and LABEL messages sent by each algorithm. The curve for M-DPOP shows the total size of the UTIL messages required for solving the main and the marginal economies. As explained in Chapter 11, M-DPOP can reuse some computation from the main economy while computing the marginal economies; the curve corresponding to M-DPOP from Figure 12.7 takes this fact into account.

R-M-DPOP spends the same amount of effort as M-DPOP for the main and marginal economies. However, R-M-DPOP also has to perform all the required LABEL propagations, which can increase the complexity by a linear factor in the worst case: one full LABEL propagation for each candidate agent, both in the main economy and in the corresponding marginal one. Figure 12.7 confirms this fact, and clearly shows that R-M-DPOP spends much more effort than M-DPOP [18]: for the largest problem size, we have a 100-fold increase, due to the LABEL propagation.[19]

In contrast, BB-M-DPOP does not incur the computational overhead introduced by the LABEL propagations. Figure 12.7 clearly shows that BB-M-DPOP requires less effort than R-M-DPOP, and is relatively close to M-DPOP. For the largest problem size, BB-M-DPOP spends just 36% more effort

[18] Notice the log scale in Figure 12.7

[19] It would, in principle, be possible to extend the LABEL propagation with the same ”safe-reusability“ principle as M-DPOP extends simple-MDPOP: we could reuse effort spent in the LABEL propagation in the main economy while performing LABEL propagation in a marginal economy. However, the current implementation and the results in Figure 12.7 do not take advantage of this possibility.

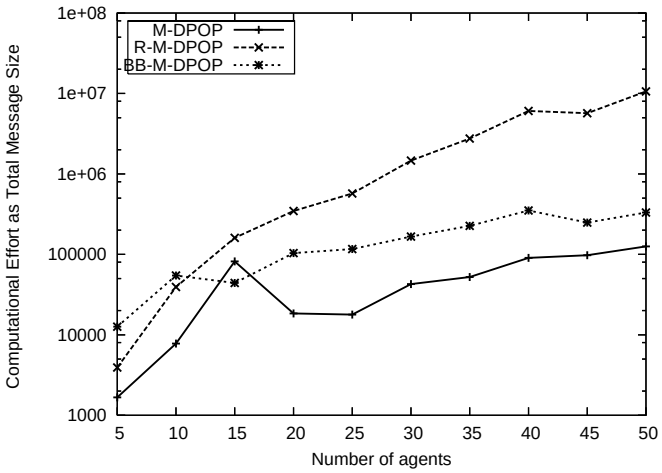


Figure 12.7: *Computational effort required by M-DPOP, R-M-DPOP and BB-M-DPOP, measured as the total size of the UTIL/LABEL messages sent.*

than M-DPOP: 150K as opposed to 110K.

12.4 Discussions and Future Work

In this section we discuss several aspects of the redistribution schemes, and point out some directions for future work.

12.4.1 Distributed implementations: incentive issues

The execution of the redistribution schemes is somewhat sensitive to the DFS tree chosen as a communication structure. Specifically, in R-M-DPOP, the choice of the DFS tree can influence which agents are eligible to be considered for receiving tax reimbursements from some other agents. Therefore, the agents may have an interest in influencing the creation of the DFS tree such that they become eligible for more reimbursements, or more "interesting" ones.

In BB-M-DPOP, the DFS tree plays a role in determining the areas where each agent is allowed to exercise its influence (e.g. in Figure 12.6, the subtree rooted at Z). Depending on how the tree is constructed, this subtree may or may not contain some variables on which an agent may have a special interest, and therefore, the DFS construction is susceptible to manipulation.

To prevent these possible manipulations, we can require a trusted third party, which will provide the agents with the DFS structure they should use. If the agents cannot influence the DFS, then the LABEL propagation from R-M-DPOP and the marginal propagations from BB-M-DPOP are faithful and give the expected results. For future work, we will investigate more elaborated versions of the LABEL and

the marginal propagations that would not be sensitive to the DFS structure used, and thus make it not profitable for agents to manipulate the DFS construction.

12.4.2 Alternate Scheme: Retaining Optimality While Returning Micropayments

Notice that it could also be possible to do the redistribution at a *micro level* so that a candidate agent $A_l \notin \{A_i, A_j\}$ is selected for each $tax_{r_j^k}(A_i)$ for each $r_j^k \in Tax(A_i)$ for each A_i, and agent A_l receives this *micropayment* if it could have had no possible influence on the payment.

The reason to consider the redistribution of micropayments is that it offers more fine-grained control over payment redistribution than when finding an agent that is eligible to receive the entire tax payment made by some agent A_i, as in R-M-DPOP. Micropayment redistribution is less brittle, in that whether or not we can redistribute payments is less sensitive to one bad choice of candidate agent A_l in R-M-DPOP. Instead, for a given VCG payment, we seek to redistribute all its component micropayments, and thus stand a better chance of being able to redistribute as much as possible from the total VCG payment.

However, care must be taken as some micropayments may be negative. To understand why, consider two agents A_i and A_j with similar preferences. Together, they are able to impose their most preferred value on a variable X, but not when taken individually. In this way, each agent can have a *positive influence* on the other one, which in turn, makes the respective VCG micropayments negative, i.e. agent A_i receives a payment for the effect on the relation of agent A_j involving this variable. Thus, we see an interesting phenomenon: while total tax payments, $Tax(A_i)$, made by every A_i are non-negative by the property of no positive externalities, an agent's presence can nevertheless have a positive externality on any *one* other agent considered in isolation, and in particular on any micropayments.

Redistributing such negative micropayments must be avoided, as it amounts to having the recipient paying taxes for some other agent and could break individual-rationality. Furthermore, by redistributing positive micropayments but not negative micropayments we can stand the risk that the redistribution will leave the bank with a budget deficit.

12.4.3 Tuning the redistribution schemes

Both in R-M-DPOP and in BB-M-DPOP, the designer can tune the execution of the algorithms and influence their performance in several ways.

R-M-DPOP: the choice of A_l can influence both the computational effort required for the $LABEL^l$ propagation, and the likelihood of finding a good recipient for the taxes, and thus the final net overall utility. In our implementation, we try to choose for each tax a recipient agent A_l which lies "as far

as possible“ from the origin of the tax, i.e. an agent A_l in a different branch of the DFS tree. This is designed to maximize the chance that A_l has no influence on the area where the tax originates.

BB-M-DPOP: here, the designer can consider different options for choosing both A_l (the recipient agent) and Z (the cutoff point). The choice of Z may have an impact on the quality of the solution chosen because the closer Z is to A_l, the less of a chance A_l has to influence its neighboring variables according to its preferences, thus decreasing the overall solution quality;on the other hand, if Z is chosen arbitrarily far then no redistribution would be possible. Therefore, in our implementation, we have tried to select an A_l as far away as possible from the area where the payments originate. Subsequently, we choose cutoff points Z which are as far as possible from A_l.

It is interesting to note that in addition to their impact on efficiency, these design choices may also have certain implications on the “fairness” of the process: which agents are considered for receiving which payments, etc. We acknowledge the importance of these issues, and will elaborate on them in future work.

12.5 Summary

We presented two methods for dealing with the VCG surplus in social choice problems when agents are self interested and have private, arbitrary utilities for different outcomes. Our algorithms are faithful, in the sense that no agent can improve its utility either by misreporting its local information or deviating from any aspect of the algorithm. The first method (R-M-DPOP) produces optimal solutions, but can only achieve a limited redistribution of the VCG payments. Our experiments show that a significant percentage of the VCG payments can be returned to the agents (close to 70%) in problems that exhibit local structure.

The second method (BB-M-DPOP) offers no optimality guarantees, but enforces full budget balance. Experiments show that both R-M-DPOP and BB-M-DPOP dominate the classical VCG mechanism in terms of the net utility of the agents, with BB-M-DPOP slightly outperforming R-M-DPOP. Experimental results show that BB-M-DPOP also requires less computational effort than R-M-DPOP. This suggests that in settings with self-interested agents where the net utility is more important than the optimality of the solution, BB-M-DPOP is the method of choice.

A very interesting avenue for future research is to investigate mechanisms that seek to redistribute *micropayments* as opposed to aggregate VCG payments (see the discussion in Section 12.4.2). Such a scheme would offer more fine-grained control over payment redistribution and would be less brittle, in that whether or not we can redistribute payments is less sensitive to one bad choice of candidate agent A_l.

In both R-M-DPOP and BB-M-DPOP, we select candidate agents to receive payments originating from parts of the problem they cannot influence. This selection has an impact on the ”quality“ of the

redistribution scheme. Specifically, in R-M-DPOP we can seek to select as a candidate agent, an agent that is very likely to be proven non-influential by the LABEL propagation scheme, and thus increase the chances of redistribution [20] In BB-M-DPOP, the question is how to choose a candidate agent such that it is likely it will have a weak influence on the tax (or not at all), and once the agent chosen, how to determine where to cut its influence such that the overall optimality is as little affected as possible.

For future work, one could also investigate more elaborated versions of the LABEL and the marginal propagations that would not be sensitive to the DFS structure used, and thus make it not profitable for agents to manipulate the DFS construction.

[20] One must take care of incentive issues, as all agents have the interest to influence this selection.

Chapter 13

Conclusions

This dissertation tackles Distributed Constraint Optimization Problems, with a particular focus on developing new efficient algorithms that make good use of the computational / memory / network resources available. In this context, previous work has concentrated mostly on adapting techniques from centralized CSP to the distributed case. In centralized CSP, search algorithms are preferred because they are fast and require small amounts of memory. Additional techniques like dynamic variable ordering, consistency maintenance, or the branch and bound principle are very successful, and further improve performance, sometimes quite impressively.

However, in a distributed setting, the conditions in which these algorithms operate are radically different. An assignment of a value to a variable is no longer instantaneously known to all agents involved, and has to be *communicated*. "The best solution found so far", or "the best cost so far" are no longer available as in centralized branch and bound, and have to be *broadcast* to all agents. By its very nature, *search* works by sequentially exploring the search space with rapid state changes, which implies many changes of context, which translates into many messages. In optimization, the search space is exponential, and thus oftentimes an exponential number of messages have to be exchanged. Techniques like dynamic reordering, or consistency maintenance from centralized CSP have been adapted to the distributed case, and oftentimes show performance improvements[26, 137, 139, 199, 200, 202, 242]. However, even with these improvements, the number of messages required is typically still very large, which implies the associated overhead is prohibitive for practical applications (Section A.1).

On the other hand, dynamic programming works by exploring the search space in a more parallel fashion: each agent computes all the possible impacts of a set of other agents on itself, and sends these valuations at once, *in a single message*. Messages are larger, but since they are fewer, the massive networking overhead associated with many small messages is avoided. Furthermore, if problem structure is taken into account (like for example by operating on a DFS tree), the maximal message size can be limited to *exponential in the induced width* as opposed to exponential in the size of the problem. Together with the fact that practical distributed problems tend to have low width, this realization is at

the core of the success of the DPOP algorithm.

In the following we present in Section 13.1 a list of the major contributions of this dissertation, and we conclude with some final remarks in Section 13.2.

13.1 Contributions

We present in the following a condensed list of the contributions of this thesis, and we continue afterwards with a more detailed view on the most important ones.

1. DPOP algorithm: distributed dynamic programming, produces a linear number of messages. Largest message exponential in the induced width of the chosen DFS. The algorithm of choice for DCOPs with low induced width.
2. DPOP extensions for efficiency (Part III):
   - a generic framework for identifying difficult areas (high-width clusters) based on problem structure (Section 6.2).
   - H-DPOP: (Chapter 5): uses consistency techniques from search to reduce message size. Can be applied in combination with most DPOP variants.
   - MB-DPOP: tradeoff between number of messages and memory/message size (Chapter 6)
   - O-DPOP: hybrid of dynamic programming and best first search that trades exponential message size for number of messages (Section 6.4)
   - LS-DPOP: Configurable large neighborhood search combined with dynamic programming (Section 7.1)
   - A-DPOP: parametrized approximation scheme, which adapts the size of the larges message to the desired approximation ratio (Section 7.2)
   - PC-DPOP: configurable centralization of high width subproblems in cluster roots, which solve them in a centralized way and integrate results into DPOP (Section 8)
3. Dynamic problem solving (Part IV):
   - SS-DPOP: self stabilizing dynamic programming (Section 9.2)
   - RS-DPOP: continuous problem solving (Section 10.3)
   - structural methods for reusing computation upon dynamic changes
   - cost-based solution stability
4. DPOP extensions for self interested agents (Part V):
   - M-DPOP: first faithful distributed mechanism for social choice (Chapter 11)

(a) implements the VCG mechanism distributedly (just a bank required)

(b) allows reuse from main to marginal economies

- Structural techniques for budget balance ((Chapter 12))

(a) R-M-DPOP: uses structure to *detect* possible influence $\rightarrow$ burns tax

(b) BB-M-DPOP: uses structure to *cut* influence $\rightarrow$ redistributes tax

In distributed constraint reasoning, several search algorithms[96, 139, 141, 197, 223, 224, 228] have been proposed. While most of these algorithms have the advantage that they can operate asynchronously and with low memory requirements, they all suffer from the problems associated with search in a distributed environment: large networking overheads caused by sending many small packets, and large algorithmic overheads due to the obligation of attaching full context information to each message because of asynchrony.

One of the most important contributions of this thesis is the dynamic programming algorithm DPOP (Chapter 4). DPOP groups many individual valuations in a single message, and it requires only a linear number of messages, thus generating low communication overheads. DPOP's complexity is given by the size of the largest UTIL message it produces, which is exponential in the induced width of the DFS ordering used. This makes DPOP very well suited for large but loose problems, which exhibit low induced width.

For problems with high induced width, however, DPOP's memory requirements may be prohibitive. In some situations, hard constraints can be exploited by methods like H-DPOP to effectively reduce message size by pruning incompatible tuples (Chapter 5).

The whole Part III of this thesis is dedicated to exploring various efficiency-related tradeoffs one can make for problems with high width, along four different dimensions: solution quality (complete vs. incomplete algorithms), memory requirements (linear / polynomial / exponential), communication requirements (few large messages vs. many small messages) and the degree of distribution (fully distributed algorithms vs. partial centralization algorithms). Several new algorithms are introduced. Table 13.1 presents a comparative overview of the current DCOP landscape. Existing algorithms are shown side by side with the new algorithms developed in this thesis (the latter ones are shown in bold). We classify all algorithms according to their memory requirements, and the number of messages they exchange, as these are the two most commonly used performance metrics. The DPOP algorithm (lower left corner) was the first in a series of algorithms exploring dynamic programming approaches in a DCOP context. Subsequently, we have developed many different extensions: typically hybrids of dynamic programming and other techniques, seeking to mitigate the exponential memory problem of DPOP by offering different tradeoffs.

[1] PC-DPOP is optimal, but sacrifices the initial distribution of the problem by partially centralizing subproblems

[2] A-DPOP is an approximation scheme, and thus sacrifices optimality

[3] LS-DPOP is a local search scheme, and thus sacrifices optimality

[4] OptAPO is optimal, but sacrifices the initial distribution of the problem by partially centralizing subproblems

Memory	Number of Messages			
	linear	polynomial	worst case exponential	exponential
linear	**PC-DPOP(1)**[1] **A-DPOP(1)**[2]	**LS-DPOP(1)**[3]	ADOPT, NCBB, AFB, SynchBB	**MB-DPOP(1)**
polynomial	**PC-DPOP(k)**[1] **A-DPOP(k)**[2]	**LS-DPOP(k)**[3]	NCBB(k), OptAPO[4]	**MB-DPOP(k)**
w.c. expon	**H-DPOP**		**O-DPOP**	
exponential	**DPOP**			

Table 13.1: Comparative overview of DCOP algorithms: memory vs. number of messages

We summarize these results in the following: Section 6.3 introduces the MB-DPOP algorithm, which can operate with bounded memory using the idea of *cycle-cuts*[51]. Section 6.4 introduces the O-DPOP algorithm, which can be applied to *open optimization problems*[70], i.e. problems that feature unbounded domains. Section 5 introduces the H-DPOP algorithm, which takes advantage of Constraint Decision Diagrams[34] (CDDs) to prune out from the UTIL messages combinations which are infeasible due to hard constraints. Section 8 introduces the PC-DPOP algorithm, which allows for the partial centralization of difficult subproblems. Section 7.1 introduces the LS-DPOP algorithm, a hybrid algorithm which is a mixture of local search and dynamic programming. Section 7.2 introduces the A-DPOP algorithm, an approximation scheme which offers a tradeoff between (guaranteed) solution quality, and computational effort.

For dynamic, distributed problems, we propose in Part IV two self stabilizing algorithms that can cope with dynamically changing problems. Different techniques for fault containment and superstabilization are presented. We also introduce a cost-based version of solution stability, and an algorithm that enforces it.

In an orthogonal area of research, we tackle the problem of dealing with strategic behavior in systems with *self-interested agents*. The issue is that existing DCOP algorithms can be manipulated by self-interested agents such that the chosen solution is no longer optimal, but better fits their interests. This is a major limitation shared by all previous DCOP algorithms. We introduce *M-DPOP*, the first *faithful* DCOP algorithm that makes honest behavior an ex-post Nash equilibrium. M-DPOP carefully integrates the Vickrey-Clarke-Groves (VCG) mechanism with DPOP. M-DPOP introduces a novel method that leverages structure in the problem to selectively *reuse computation* performed in solving the main problem while solving the marginal problems, in a way that is robust against manipulation by the excluded agents. We have also introduced two extensions to M-DPOP (see Chapter 12) that address the inefficiency of the VCG mechanism that taxes must be burned, thus creating a welfare loss to the agents. Our extensions exploit structure in the problem to develop faithful methods to redistribute payments back to agents, reducing this cost on the system.

All the algorithms described in this thesis operate in an essentially synchronized fashion. While this avoids the inconvenience of overhead (see Section A.2), asynchronous techniques have potential advantages in terms of ability to deal with message loss and/or slow links, and sometimes they offer the possibility of *anytime solving* (i.e. offering some solution fast, and then improving it as time goes by). Nevertheless, anytime behavior can also be obtained from dynamic programming algorithms, like for example using the iterative versions of A-DPOP (Section 7.2.6), or LS-DPOP (Section 7.1.3). One could also envisage an online version of the O-DPOP algorithm, where partial results are sent upstream before proving their optimality.

While in this thesis we have not dealt with message loss explicitly [1] the self-stabilizing algorithms in Chapters 9 and 10 handle this problem by simply resending the lost messages. An alternative is provided by the AnyPOP algorithm from Section 7.2.5, which deals with messages that have not yet arrived by considering *what could have been their influence*, based on the messages that did arrive, and the local relations.

13.2 Concluding Remarks

In this thesis, we have investigated Distributed Constraint Optimization Problems as an approach to effective coordination in multiagent systems. This is an important topic because DCOPs are applicable to many real life problems that are distributed by nature.

Among the most challenging issues in DCOP is *efficiency*. For problems of practical interest, current search-based algorithms are too inefficient to be used in a realistic application. The main issue is that typically they require exponential numbers of small messages, which in turn produces enormous networking overheads and delays. We have proposed in this thesis a class of algorithms based on dynamic programming which address this issue by (a) discovering problem structure by using DFS trees, and exploiting it when possible, and (b) by packaging together many valuations in larger messages which can be transported over the network more efficiently, with less overhead. The resulting algorithms have been shown by experimental results to be up to 5 orders of magnitude more efficient than search based algorithms. We therefore believe that whenever memory constraints allow for algorithms based on dynamic programming, such algorithms are preferable to search based algorithms. In situations where this is not possible because of excessive memory requirements, different hybrid algorithms can be tried, as discussed in Part III of this thesis.

An important feature of DCOPs is that often times agents operate in open and dynamic environments, with agents *dynamically* joining or leaving the system, resources appearing or being consumed, tasks being allocated and carried out, etc. Chapter 9 introduces two self-stabilizing algorithms that can operate in such dynamic, distributed environments. Chapter 10 discusses *solution stability* in dynamic

[1] The TCP underlying networking protocol deals with the problems of packet loss and out-of-order delivery, thus freeing higher level algorithms of the task of reasoning about these problems explicitly

environments, and introduces a self-stabilizing version of DPOP that maintains it.

Another key challenge in DCOPs is dealing with dishonest behavior in systems with *self-interested agents*. This poses effectively a big problem to DCOP algorithms, which can be manipulated by such self-interested agents such that the final solution discovered is better suited to themselves, regardless of the global optimum. This renders existing DCOP algorithms useless in such settings, as the results they obtain are meaningless. Existing work to address this problem is limited at most. We introduce *MDPOP*, the first DCOP algorithm that provides a *faithful distributed implementation* for efficient social choice. Faithfulness ensures that no agent can benefit by unilaterally deviating from any aspect of the protocol, and is achieved by carefully integrating the Vickrey-Clarke-Groves (VCG) mechanism with DPOP. M-DPOP introduces a novel method that leverages structure in the problem to selectively *reuse computation* performed in solving the main problem while solving the marginal problems, in a way that is robust against manipulation by the excluded agents. We have also introduced two extensions to M-DPOP (see Chapter 12) that address the inefficiency of the VCG mechanism that taxes must be burned, thus creating a welfare loss to the agents. Our extensions exploit structure in the problem to develop faithful methods to redistribute payments back to agents, reducing this cost on the system.

Appendix A

Appendix

A.1 Performance Evaluation for DCOP algorithms

Performance evaluation of distributed constraint reasoning algorithms is a long-debated subject. Several metrics have been proposed so far:

- *number of messages*, or "network load"[123] required by an algorithm to find the (optimal) solution. This metric is meaningful when all algorithms compared produce messages of comparable sizes. This is not the case when comparing DPOP with search algorithms, for example.

- number of *constraint checks (CC)* performed while solving the problem. This is a metric heavily used in *centralized* CSP, which offers the advantage that it is independent of the algorithm used and of the hardware platform.

- number of *simulator cycles*[228]. This assumes a simulator is used, and in each *synchronous cycle*, each agent reads its incoming messages, performs computation, and sends out its messages for delivery in the next cycle. The metric counts the number of such cycles performed while solving the problem.

- the *longest sequence of messages (LSM)*: this is the DisCSP equivalent of Lamport's logical clock[117], and is a measure of the *duration* of the execution of the algorithm.

- number of *concurrent constraint checks (CCC)* performed by the agents while solving the problem. This is an adaptation of the *cc* metric from centralized CSP to the distributed case, and measures the *computation* performed by the agents (in parallel).

- number of *non concurrent constraint checks (NCCC)* performed by the agents while solving the problem. This is an adaptation of the *cc* metric from centralized CSP to the distributed case.

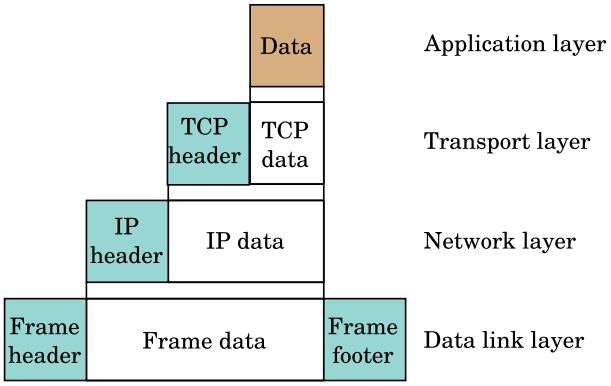


Figure A.1: *Encapsulation of a TCP packet.*

Most of the metrics devised so far were designed for comparing very similar algorithms, all of which were based on search. Therefore, all the algorithms were producing messages of comparable size (linear in the number of variables), and thus the "number of messages" metric was adequate. However, with the introduction of DPOP, it is clear that this metric by itself is no longer sufficient, at least not when comparing DPOP against a search-based algorithm. Specifically, consider that some UTIL messages in DPOP can contain millions of valuations! Such a "logical" message obviously cannot count as a single message while comparing DPOP with a search algorithm. All "logical" UTIL messages are subject to possible fragmentation into multiple smaller messages by the lower networking layer, and DPOP must be penalized in such cases. To our knowledge, so far the lower networking layers have been ignored while devising performance metrics. We believe that in order to ever deploy a DisCSP/DCOP system in a real environment, we need to consider also the details of the underlying network, and understand its behavior, strengths and limitations.

Let us consider the following scenario: the "agents" in our DCOP are real people, each one with their own computer connected to the internet, and a (complex) local problem. Local problems are connected with other agents' local problems, and the neighboring agents are expected to be able to communicate with each other. Different connectivity scenarios are possible:

- users in a company, connected to the company LAN. These are fast connections, 100Mbit or even Gigabit. Latency is typically around 10ms[1].
- home users, or small companies, connected to internet via a broadband connection. These are (relatively) fast connections, 256Kbit or more [1]. Latency is above 100ms, typically between 150ms and 200ms[1].
- large industrial users, connected to backbones via fiber optic.

[1] At the time of this writing, my broadband connection is 4Mbit

In the following, we assume that the TCP/IP protocol stack is used by the agents to communicate. A TCP packet is encapsulated by the physical layer in (ethernet) *frames* (as shown in Figure A.1) composed of:

- MAC header: 14 bytes + 4 bytes CRC final field (overhead). Minimal size of an Ethernet frame is 64 bytes, out of which the data is at least 46 bytes (if less, then padded with 0's)
- IP header: size 20 bytes (overhead).
- TCP header: size: 20 bytes (overhead)
- TCP payload: typically 1000-1500 bytes.

The following parameters of the network are of interest (to simplify our analysis, we assume they hold throughout the network of agents, without variations):

- L: latency for one packet: this is the time it takes for one packet to travel to its destination. Typical latency for local LAN: 10ms[1]. Typical latency for Internet: above 100ms; typically between 150ms and 200ms[1].
- B: communication bandwidth: the rate at which data can be sent over a connection. Typical for LAN is 10Mbit, 100Mbit, Gigabit (nowadays mostly 100Mbit, Gigabit is quite common). Wireless (11,54,108Mbps). For Internet, slowest links are 56kbps.
- N_o: Networking overhead per packet, in bytes (size of MAC headers, TCP/IP headers)
- $TCP\_payload$: size of payload in a TCP message, in bytes (typically 1000-1500 bytes). If a message contains more than $TCP\_payload$ bytes, it will be split in several messages. Networking overhead is incurred for each resulting message.

In addition to the characteristics of the network, each algorithm has its own specifics:

- *algorithm-introduced communication overhead*: for each message m_i, $A_o(m_i)$: size of algorithm required context information (in ADOPT, the current view; in DPOP, the IDs and domain sizes of variables in the message).
- *payload per message*: for a message m_i, $Payload(m_i)$ is the size of the useful information the message contains. In ADOPT, this is always 1 (one cost value reported). In DPOP, this is exponential in the number of dimensions of the message.

To capture the characteristics of the underlying transport protocol, we propose an adaptation of the LSM metric, which (a) takes into account message size, thus penalizing DPOP for sending large messages, and (b) takes into account the characteristics of the lower transport layers:

Definition 47 (Network-based LSM (NLSM)) *Considering the Longest Sequence of Messages metric (LSM), we define the Network-based LSM (NLSM) metric as follows:*

$$NLSM = \sum_{\forall m_i \in LSM} \left\lceil \frac{(Payload(m_i)+A_o(m_i))}{TCP\_payload} \right\rceil$$

Notice that in the sum, only the messages m_i that participated in the LSM are considered. Thus, NBR effectively accounts for the longest message passing sequence, while at the same time considering large messages as splitted across multiple, smaller, packets.

Furthermore, we argue that in any practical deployment of a DCOP application on a real network, an important performance measure is the *runtime (in seconds)* of the algorithm until the solution is found, *given the characteristics of the network*. To capture this runtime, we propose to use NLSM (which already takes into account the transport layer, i.e. TCP), and to adapt NLSM to account for the physical network where the algorithm is deployed:

Definition 48 (Network Based Runtime (NBR)) *Considering NLSM, we define the Network Based Runtime (NBR) metric as follows:*

$$NBR = NLSM \times L$$

NBR thus gives a measure of the total time spent by an algorithm to solve a problem, on a particular network with the given latency L.

Note that NBR makes a number of simplifying assumptions: the latency L holds throuhout the network, for the entire execution of the algorithm, and for all algorithms measured, i.e. no significant variations in latency related to geographical position, time, or algorithm. While we acknowledge that these assumptions are debatable in a real deployment of a DCOP algorithm, we believe they are reasonable, and that NBR is a more realistic metric than the ones previously proposed for DCOP.

Additionally, to measure the degree to what a DCOP algorithm makes appropriate use of network bandwidth, we define the

Definition 49 (Communication Overhead) *The communication overhead is the total amount of information (bytes) which is not essential to the algorithm, but sent over the network nevertheless: Overhead = $\sum_{\forall m_i} Overhead(m_i) = \sum_{\forall m_i} N_o(m_i) + A_o(m_i)$;*

Notice that this sum is defined over all messages sent over the network, not just LSM as in Definition 48.

Example 27 (Message Passing Example) *Consider the example problem from Figure 3.3. Consider agent X_{11}, who sends messages to its parent X_5. For simplicity, assume all variables have 10 values*

in their domain. Assume that the range of possible valuations is between 0 and 255, for the largest aggregated valuation. Thus, all valuations can be encoded as one byte.

***ADOPT**: X_{11} sends COST messages of this form: $COST(X_0 = 1, X_2 = 2, X_5 = 4) = 4$, i.e. "in the context of $X_0 = 1, X_2 = 2, X_5 = 7$, the cost for X_{11} is 7. An economic encoding of such a message requires at least 6 bytes: 3 bytes for the IDs of the variables X_0, X_2, X_5, and 3 for their current values. The useful payload of this message is just the cost value 7 (i.e. 1 byte). The algorithm overhead is $A_o = 6$ bytes.*

The message has to be sent over the network. Assume the best case: the COST message is sent using a single TCP message, which uses a single Ethernet frame. The minimal size of such a frame is 64 bytes: MAC headers of 14 bytes, plus a minimum of 46 bytes of TCP/IP payload data (if less, padded with 0's), plus 4 bytes CRC. The overhead introduced by the networking layer is $N_o = 64 - A_o - Payload = 64-6-1 = 57$ bytes. Total overhead is $Overhead = A_o+N_o = 63$ bytes. Overhead to payload ratio: 63:1. To simplify the analysis, assume ADOPT does not manage to perform any pruning in this case, and the whole set of 1000 combinations of assignments for X_0, X_2, X_5 will be explored. Therefore, X_{11} will receive at least 1000 VALUE messages of the form $X_0 = 1, X_2 = 2, X_5 = 7$, and will reply with 1000 individual *COST messages of the form $COST(X_0 = 1, X_2 = 2, X_5 = 4) = 4$. This implies that ADOPT requires 63,000 bytes of useless information sent for 1000 bytes of payload.*

***DPOP**: the UTIL message $UTIL_{11}^5$ sent from X_{11} to X_5 contains $10^3 = 1000$ valuations, one for each combination of values of variables X_5, X_2, X_0. The message has a header containing the list of the variables involved (i.e. X_5, X_2, X_0), and the size of their domains (i.e. 10,10,10). For the example above, this requires 6 bytes, thus $A_o = 6$ bytes. The 1000 valuations are simply included in the message as a sequence of 1000 bytes, which typically* fit into a single message. *The networking overhead for this message is then $N_o = 58$ bytes (MAC header, IP header, TCP header, MAC final CRC field) Therefore, a UTIL message of 1000 valuations has total size 1064 bytes, and total overhead $Overhead = N_o + A_o = 58 + 6 = 64$ bytes. Overhead to payload ratio: 64:1000.*

***DPOP splitting large messages**: assume DPOP has to send a message with 4 dimensions (i.e. 10,000 values). Considering the typical payload of a TCP message of 1000 bytes, it follows that the UTIL message will have to be split into 10 TCP messages. Now the message contains 4 dimensions, therefore we count 8 bytes for the header describing the variables and their domains. $Overhead = A_o + 10 \times N_o = 8 + 580 = 588$ bytes for 10,000 valuations (the algorithmic overhead A_o is counted only once)*

In contrast, ADOPT sending the same amount of information would incur an $Overhead = 10000 \times (A_o + N_o) = 10,000 \times (8 + 55) = 630,000$ bytes of overhead when sending 10,000 valuations.

Assuming equal latencies of 100ms for a fast connection over the internet, and assuming all the 1000 valuations are sent sequentially by ADOPT, we have:

$NBR_{ADOPT} = 10000 \times L = 1000s$.

$NBR_{DPOP} = 10 \times L = 1s.$

A.2 Performance Issues with Asynchronous Search

Asynchronous search algorithms with polynomial memory bounds have the advantage that they allow the agents to operate asynchronously, and have low memory requirements. However, they have a series of drawbacks, that we outline in the following.

Issues with search algorithms in general:

- In general, in order to be able to guarantee polynomial memory requirements, full caching [8, 32, 42, 132, 170] is not possible (see Section 3.1.1.3). In such cases, re-exploration of parts of the search space may be required [33, 141, 170, 240]. This means that even after the whole search space has been explored and the *cost* of the best solution has been found, the algorithm has to re-explore parts of the search space again to actually derive the solution itself. This implies even more work than necessary for the agents in terms of computation, and more network load in terms of message passing.

- search algorithms introduce significant networking communication overheads by the fact that they use many small messages, which contain as payload just a single cost value, i.e. typically 1 byte (see Section A.1). If effective pruning is not possible, the overhead becomes prohibitive.

Additionally, *asynchronous* search algorithms introduce the following performance issues:

- asynchronous algorithms produce significant algorithmic communication overheads by the fact that due to their asynchrony, they have to attach *context* information to each message (see Section A.1).

- random delays in message delivery (which are the norm in any realistic network) sometimes significantly degrade their performance, both in terms of computation and message passing [240, 241, 243].

A.3 FRODO simulation platform

We have developed and released a ”FRamework for Open/Distributed Optimization” (FRODO), that simulates in a single Java virtual machine a multiagent platform geared towards the implementation and testing of (distributed) optimization algorithms. Each agent is simulated with a Java thread, and communicates with its peers via message exchange.

In Figure A.2 we present an overview of the architecture of the platform. Briefly, there is an environment that is responsible for creating the agent threads and message delivery. Within the environment, each agent operates in an autonomous fashion: it loads its relevant subproblem, and then participates in a message exchange protocol with (some of) its peers, as dictated by the optimization algorithm.

The environment can monitor the message exchange, and can present a GUI to the user that shows the current state of the solving process. For example, in the resource allocation example in the sensor network environment, the GUI shows the current allocations of sensors to targets, and the conflicts that are still to be resolved. For more details, and screenshots of the simulator, please visit `http://liawww.epfl.ch/Research/sensornets/`.

In the public version there are two implemented algorithms: Distributed Breakout Algorithm - DBA[228], and DPOP[160]. The framework is extensible, and allows for easy implementation and testing of new optimization algorithms, be they centralized or distributed.

There are also available two testbeds: one for resource allocation in a sensor network, and one for meeting scheduling problems. Both have random problem generators, and GUIs to display the problem instances.

More details, documentation, paper and source download can be found in[154] and at

`http://liawww.epfl.ch/frodo/`.

A.4 Other applications of DCOP techniques

A.4.1 Distributed Control

In a highway network, many problems like traffic jams or accidents can be avoided with more effective and adaptive speed limitations. Such adaptive control can be provided by intelligent agents, each one responsible for a highway segment. Neighboring agents can communicate with each other to exchange information about traffic conditions, enforced speed limits, etc. The objective is to make the traffic a fluid as possible, and increase safety.

We have developed a DCOP model of this problem[175], where the agents correspond to highway segments and they control the speed limitation for their respective segments. Constraints between neighboring agents are designed to model safety restrictions (e.g. enforcing a speed limit of 60 km/h on a segment immediately after a segment with 120 km/h is dangerous), and to increase throughput.

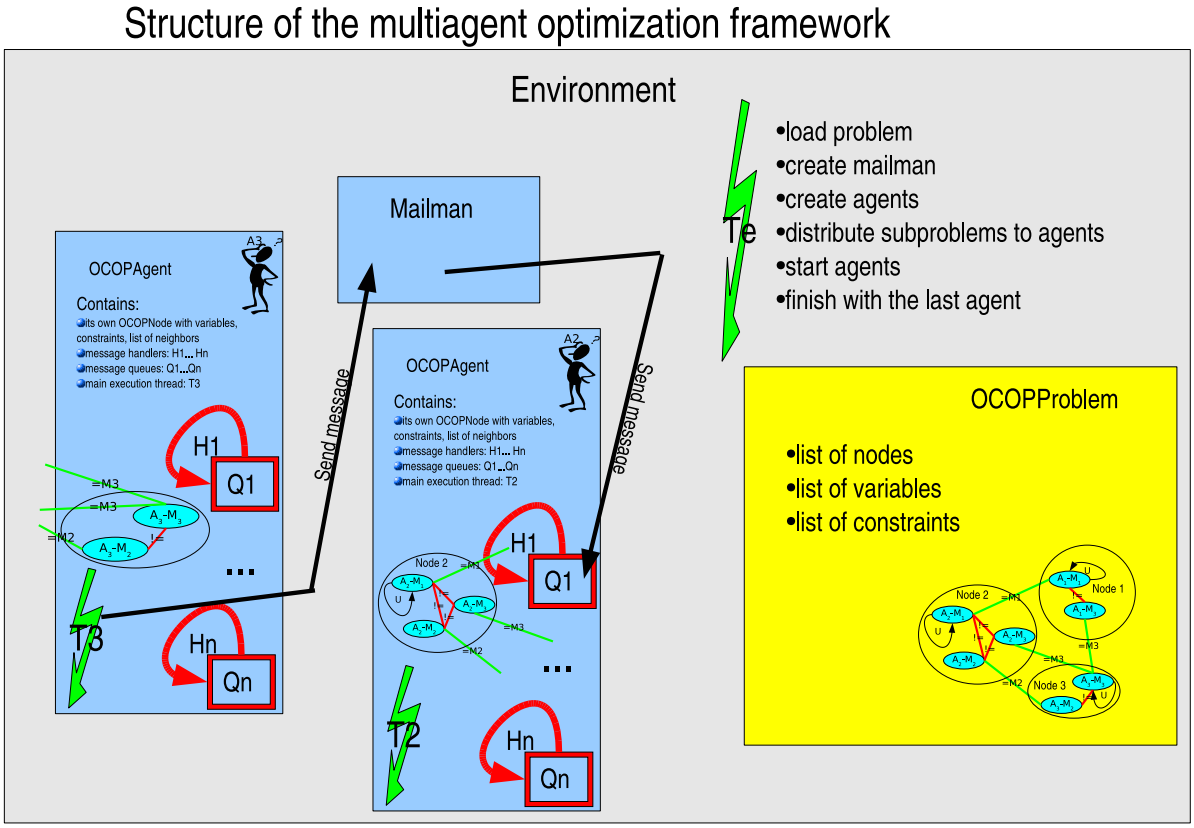


Figure A.2: *FRODO platform: agents simulated as threads that exchange messages with their peers. New optimization algorithms can be easily implemented and tested. The environment provides monitoring and visualization support.*

A.4.2 Distributed Coordination of Robot Teams

Cooperative robotics is an area where multiple autonomous agents often have to accomplish a common goal, such as finding an object, moving an object, patrolling, etc. Often times, the goal is too complex for each one of the individual robots to achieve by itself: the area to patrol may be too large for a single robot, the object to move may be too heavy, etc. In such settings, the robots have to cooperate in order to achieve the goal, and effective coordination is essential.

In[97] we investigate a scenario where a team of robots must find an odor source as fast as possible. They have sensors for odor and for the wind direction on board, and can track the odor source by reasoning about the direction of the wind, and about the possible location of the source. Team work can lead to finding the source much faster than a single robot could do, but requires effective coordination among the robots. Modeling the coordination problem as a DCOP and executing a variant of DPOP to solve it dynamically as the robot teams evolve in the environment can lead to significant improvements in terms of the time required to find the source, and of the total effort spent by the robots to find the

source.

A.5 Relationships with author's own previous work

Parts of this thesis have appeared as preliminary versions in the following publications:

- Optimization algorithms:
  1. the DPOP algorithm (Chapter 4) appears in[160]
  2. the H-DPOP algorithm (Chapter 5) appears in[114]
  3. the MB-DPOP algorithm (Section 6.3) appears in[167]. An early version appears in[156]
  4. the O-DPOP algorithm (Section 6.4) appears in[168]
  5. the LS-DPOP algorithm (Section 7.1) appears in[163]
  6. the A-DPOP algorithm (Section 7.2) appears in[158], and an extended version in[159]
  7. the PC-DPOP algorithm (Chapter 8) appears in[169]
  8. an improvement to the DBA algorithm[226] using interchangeabilities[77] appears in[155]. Another improvement of DBA consisting in a value-ordering heuristic appears in[157]
- Dynamic Systems:
  1. the S-DPOP algorithm (Chapter 9) appears in[165]
  2. the RS-DPOP algorithm and solution stability (Chapter 10) appear in[164]
- Self-interested agents:
  1. the M-DPOP algorithm (Chapter 11) appears in[171]. An early version appears in[162]
  2. the BB-M-DPOP algorithm (Chapter 12) appears in[171, 172]
- Privacy:
  1. a secure version of the DPOP algorithm using multiparty computation appears in[196]
  2. an efficient, secure version of the DPOP algorithm appears in[69]
- Applications:
  1. an application to distributed scheduling of preventative maintenance of generating units in a power plant appears in[166]
  2. applications to distributed meeting scheduling problems are discussed in[160, 161, 167–169, 171]
  3. applications to graph coloring problems are discussed in[160, 161, 167, 169]
  4. applications to sensor networks are discussed in[155, 157, 160, 161, 167, 169]

5. applications to combinatorial auctions are discussed in[69, 114]
6. distributed coordination of robot teams (Section A.4)[97]
7. distributed control (Section A.4)[175]

Bibliography

[1] Internet traffic report. http://www.internettrafficreport.com/, Aug 2007.

[2] H. H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Trans. Comput.*, 37(4):449–453, 1988.

[3] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.*, 123(1-3):75–102, 2002.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *20th ACM Symposium on Operating Systems Principles*, 2005.

[5] N. Andelman, M. Feldman, and Y. Mansour. Strong price of anarchy. In *ACM-SIAM Symposium on Discrete Algorithms 2007 (SODA'07)*, 2007.

[6] M. Arshad and M. C. Silaghi. Distributed simulated annealing and comparison to DSA. In *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, IOS Press, "Frontiers in Artificial Intelligence"*, 2004.

[7] C. d. Aspremont and L. A. Gerard-Varet. Incentives and incomplete information. *Journal of Public Economics*, 11:25–45, 1979.

[8] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 20–28, San Francisco, CA, 2003. Morgan Kaufmann.

[9] M. J. Bailey. The demand revealing process: To distribute the surplus. *Public Choice*, 91(2):107–26, April 1997.

[10] V. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.

[11] A. Barrett. Autonomy architectures for a constellation of spacecraft. In *Proc. of the 5th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS-99)*, pages 291–296, Noordwijk, The Netherlands, 1999.

[12] M. Basharu, I. Arana, and H. Ahriz. Solving DisCSPs with penalty driven search. In *AAAI*, pages 47–52, 2005.

[13] R. Bayardo and D. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*, Montreal, Canada, 1995.

[14] R. Bejar, C. Fernandez, M. Valls, C. Domshlak, C. Gomes, B. Selman, and B. Krishnamachari. Sensor networks and distributed CSP: Communication, computation and complexity. *Artificial Intelligence*, 161(1-2):117–147, 2005.

[15] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[16] R. E. Bellman and S. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[17] J. C. Benaloh. Secret sharing homomorphisms: keeping shares of a secret secret. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 251–260, London, UK, 1987. Springer-Verlag.

[18] R. Bent and P. V. Hentenryck. Dynamic vehicle routing with stochastic requests. In *IJCAI*, pages 1362–1363, 2003.

[19] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, Inc., Orlando, FL, USA, 1972.

[20] C. Bessière. Arc-consistency for non-binary dynamic CSPs. In *Proc. of the 10th ECAI*, pages 23–27, Vienna, Austria, 1992.

[21] C. Bessière. Arc-consistency and arc-consistency again. *Artifficial Intelligence*, 65(1):179–190, 1994.

[22] C. Bessière, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artif. Intell.*, 161(1-2):7–24, 2005.

[23] B. Bidyuk and R. Dechter. On finding minimal w-cutset. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 43–50, Arlington, Virginia, United States, 2004. AUAI Press.

[24] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[25] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Proceedings 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97, LNCS vol. 1295*, pages 19–36, Berlin, 1997. Springer-Verlag.

[26] I. Brito and P. Meseguer. Distributed forward checking. *Principles and Practice of Constraint Programming*, LNCS 2833:801–806, 2003.

[27] K. Brown and D. Fowler. Extending constraint programming with uncertain reasoning for robust dynamic scheduling. In K. Brown and C. Beck, editors, *CUW'01: Constraints and Uncertainty workshop at CP'2001*, Cyprus, November 2001.

[28] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[29] R. Cavallo. Optimal decision-making with minimal waste: Strategyproof redistribution of VCG payments. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*, Hakodate, Japan, May 2006.

[30] H. Chalupsky, Y. Gil, C. A. Knoblock, K. Lerman, J. Oh, D. V. Pynadath, T. A. Russ, and M. Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Thirteenth Conference on Innovative Applications of Artificial Intelligence Conference*, pages 51–58. AAAI Press, 2001.

[31] A. Chechetka and K. Sycara. A decentralized variable ordering method for distributed constraint optimization. Technical Report CMU-RI-TR-05-18, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[32] A. Chechetka and K. Sycara. An any-space algorithm for distributed constraint optimization. In *Proceedings of AAAI Spring Symposium on Distributed Plan and Schedule Management*, March 2006.

[33] A. Chechetka and K. Sycara. No-Commitment Branch and Bound Search for Distributed Constraint Optimization. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*, Hakodate, Japan, May 2006.

[34] K. C. K. Cheng and R. H. C. Yap. Constrained decision diagrams. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, pages 366–371, Pittsburgh, USA, 2005.

[35] T.-Y. Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans. Software Eng.*, 9(4):504–512, 1983.

[36] I. Cidon. Yet another distributed depth-first-search algorithm. *Inf. Process. Letters*, 26(6):301–305, 1988.

[37] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, 11:17–33, 1971.

[38] Z. Collin, R. Dechter, and S. Katz. On the Feasibility of Distributed Constraint Satisfaction. In *Proceedings of the 12^{th} International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 318–324, Sidney, Australia, 1991.

[39] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 1999.

[40] Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[41] W. Conen and T. Sandholm. Partial-revelation VCG mechanism for combinatorial auctions. In *Proc. 18th Nat. Conf. on Artificial Intelligence (AAAI'02)*, 2002.

[42] A. Darwiche. Recursive conditioning. *Artifficial Intelligence*, 126(1-2):5–41, 2001.

[43] R. K. Dash, N. R. Jennings, and D. C. Parkes. Computational-mechanism design: A call to arms. *IEEE Intelligent Systems*, pages 40–47, November 2003. Special Issue on Agents and Markets.

[44] J. Davin and P. J. Modi. Impact of problem centralization in distributed constraint optimization algorithms. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1057–1063, New York, NY, USA, 2005. ACM Press.

[45] E. Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.

[46] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, pages 63–109, 1983.

[47] S. de Vries and R. V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

[48] R. Dechter. Constraint Networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.

[49] R. Dechter. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints: An International Journal*, 7(2):51–55, 1997.

[50] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[51] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[52] R. Dechter and A. Dechter. Structure driven algorithms for truth maintenance. *Artificial Intelligence Journal*, 82:1–20, 1996.

[53] R. Dechter and Y. E. Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2):93–118, 2001.

[54] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 2006. To appear.

[55] R. Dechter and J. Pearl. Tree clustering schemes for constraint-processing. *Artificial Intelligence*, 3(38):353–366, April 1989. Dual graph representation.

[56] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

[57] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[58] A. H. L. A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, Jul 1960.

[59] S. Dolev. *Self-Stabilization*. MIT Press, March 2000.

[60] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[61] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO'92*, pages 139–147, 1992.

[62] C. Eisenberg and B. Faltings. Using the breakout algorithm to identify hard and unsolvable subproblems. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003), Lecture Notes in Computer Science*, Kinsale, County Cork, Ireland, September 2003.

[63] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Negotiating socially optimal allocations of resources. *Journal of Artificial Intelligence Research*, 25:315–348, 2006.

[64] E. Ephrati and J. Rosenschein. The Clarke tax as a consensus mechanism among automated agents. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-91*, pages 173–178, Anaheim, CA, July 1991.

[65] E. Ephrati and J. S. Rosenschein. Deriving consensus in multi-agent systems. *Journal of Artificial Intelligence*, 87:21–74, 1996.

[66] E. Ephrati and J. S. Rosenschein. A heuristic technique for multiagent planning. *Annals of Mathematics and Artificial Intelligence*, 20:13–67, 1997.

[67] O. Ergun and J. Orlin. Dynamic programming methodologies in very large scale neighborhood search applied to the traveling salesman problem. Technical Report 4463-03, MIT, Sloan School of Management, Dec. 2004.

[68] B. Faltings. A budget-balanced, incentive-compatible scheme for social choice. In *Workshop on Agent-mediated E-commerce (AMEC) VI*. Springer Lecture Notes in Computer Science, 2004.

[69] B. Faltings, T. Leaute, and A. Petcu. Method to allocate inter-dependent resources by a set of participants. US/PCT Patent pending, 2007.

[70] B. Faltings and S. Macho-Gonzalez. Open constraint programming. *Artificial Intelligence*, 161(1-2):181–208, January 2005.

[71] B. Faltings, D. Parkes, A. Petcu, and J. Shneidman. Optimizing streaming applications with self-interested users using M-DPOP. In *COMSOC'06: International Workshop on Computational Social Choice*, pages 206–219, Amsterdam, The Netherlands, December 2006.

[72] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, pages 173–182, 2002.

[73] J. Feigenbaum, V. Ramachandran, and M. Schapira. Incentive-compatible interdomain routing. In *Proceedings of the 7th Conference on Electronic Commerce*, pages 130–139, 2006.

[74] J. Feigenbaum and S. Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, 2002.

[75] S. Fitzpatrick and L. Meertens. An experimental assesment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *1st symposium on stochastic algorithms: foundations and applications*, pages 49–64, 2001.

[76] E. C. Freuder. Partial Constraint Satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*, pages 278–283, 1989.

[77] E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-91*, pages 227–231, Anaheim, CA, 1991.

[78] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI-85*, pages 1076–1078, Los Angeles, CA, 1985.

[79] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992.

[80] S. Fujita, S. Tagashira, C. Qiao, and M. Mito. Distributed branch-and-bound scheme for solving the winner determination problem in combinatorial auctions. In *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 661–666, Washington, DC, USA, 2005. IEEE Computer Society.

[81] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward-bounding for distributed constraints optimization. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06)*, Riva del Garda, Italy, August 2006.

[82] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[83] M. L. Ginsberg, A. J. Parkes, and A. Roy. Supermodels and robustness. In *AAAI/IAAI*, pages 334–339, 1998.

[84] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *UAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, Arlington, Virginia, United States, 2004. AUAI Press.

[85] S. W. Golomb and L. D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[86] C. Gomes, C. Fernandez, R. Bejar, and B. Krishnamachari. Communication and computation in DisCSP algorithms. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 40–45, Ithaca, NY, USA, 2002.

[87] J. Green, E. Kohlberg, and J.-J. Laffont. Partial equilibrium approach to the free-rider problem. *Public Economics*, 6:375–394, 1976.

[88] J. Green and J.-J. Laffont. On the revelation of preferences for public goods. *Public Economics*, 8:79–93, 1977.

[89] J. Green and J.-J. Laffont. Révélation des Préférences pour les Biens Publics. *Cahiers du Séminaire d'Econométrie*, 19:83–103, 1977.

[90] R. Greenstadt, J. P. Pearce, and M. Tambe. Analysis of privacy loss in distributed constraint optimization. In *Proc. of Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, 2006.

[91] T. Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[92] T. Groves and J. O. Ledyard. Some limitations of demand revealing processes. *Public Choice*, 29:107–124, 1977.

[93] T. Groves and M. Loeb. Incentives and public inputs. *J. of Public Economics*, 4:311–326, 1975.

[94] M. Guo and V. Conitzer. Worst-case optimal redistribution of VCG payments. In *Proceedings of the 8th ACM Conference on Electronic Commerce (EC-07)*, San Diego, CA, USA, 2007.

[95] Y. Hamadi. Optimal distributed arc-consistency. In *CP '99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 219–233, London, UK, 1999. Springer-Verlag.

[96] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI-98*, pages 219–223, 1998.

[97] A. Herrmann, A. Petcu, T. Lochmatter, T. Leaute, and B. Faltings. Collective search of a single odour source using cooperation. Technical report lia-report-2007-003, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), 2007.

[98] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 222–236, 1997.

[99] A. Holland and B. O'Sullivan. Weighted super solutions for constraint programs. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, 2005.

[100] R. Huebsch, J. M. Hellerstein, N. Lanham, et al. Querying the Internet with PIER. In *VLDB*, Sept. 2003.

[101] L. Hurwicz. On the existence of allocation systems whose manipulative Nash equilibria are Pareto optimal. unpublished, 1975.

[102] S. Izmalkov, S. Micali, and M. Lepinski. Rational secure computation and ideal mechanism design. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 585–595, Washington, DC, USA, 2005. IEEE Computer Society.

[103] M. O. Jackson. Mechanism theory. In *The Encyclopedia of Life Support Systems*. EOLSS Publishers, 2000.

[104] E. Kaplansky and A. Meisels. Distributed personnel scheduling - negotiation among scheduling agents. *Annals of Operations Research*, 2005. Accepted, July 2005.

[105] K. Kask and R. Dechter. A graph-based method for improving GSAT. In *AAAI/IAAI, Vol. 1*, pages 350–355, 1996.

[106] K. Kask and R. Dechter. Branch and bound with mini-bucket heuristics. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 426–435, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[107] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying cluster-tree decompositions for automated reasoning in graphical models. *Artificial Intelligence*, 2005.

[108] J. Katz and S. D. Gordon. Rational secret sharing, revisited. In *Proc. Security and Cryptography for Networks*, 2006.

[109] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[110] U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, University of Aalborg, Denmark, 1990.

[111] T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[112] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, Jan 1975.

[113] D. E. Knuth. The stanford graphbase: a platform for combinatorial algorithms. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 41 – 43, 1993.

[114] A. Kumar, A. Petcu, and B. Faltings. H-DPOP: Using hard constraints to prune the search space. In *IJCAI'07 - Distributed Constraint Reasoning workshop, DCR'07*, pages 40–55, Hyderabad, India, Jan 2007.

[115] J.-J. Laffont and E. Maskin. A differential approach to dominant strategy mechanisms. *Econometrica*, 48:1507–1520, 1980.

[116] S. M. Lahaie, F. Constantin, and D. C. Parkes. More on the power of demand queries in combinatorial auctions: Learning atomic languages and handling incentives. In *Proc. 19th Int. Joint Conf. on Artificial Intell. (IJCAI'05)*, 2005.

[117] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[118] A. H. Land and A. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[119] J. Larrosa and R. Dechter. Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints*, 8(3):303–326, 2003.

[120] J. Larrosa, K. Kask, and R. Dechter. Up and down mini-bucket: a scheme for approximating combinatorial optimization tasks. Technical report, University of California at Irvine, 2001.

[121] R. Lavi, A. Mu'alem, and N. Nisan. Towards a characterization of truthful combinatorial auctions. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, 2003.

[122] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of ACM Conference on Electronic Commerce, EC-00*, pages 235–245, 2000.

[123] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1997.

[124] A. Lysyanskaya and N. Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *26th Annual Int. Cryptology Conference (CRYPTO'06)*, 2006.

[125] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[126] R. Maheswaran, J. Pearce, E. Bowring, P. Varakantham, and M. Tambe. Privacy loss in distributed constraint reasoning: A quantitative framework for analysis and its applications. *JAAMAS*, 2006.

[127] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS-04*, 2004.

[128] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.

[129] R. Mailler and V. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research (JAIR)*, 2005.

[130] L. Makowski and J. M. Ostroy. Vickrey-Clarke-Groves mechanisms in continuum economies : Characterization and existence. *Journal of Mathematical Economics*, 21:1–35, 1992.

[131] R. Marinescu and R. Dechter. AND/OR branch-and-bound for graphical models. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, Edinburgh, Scotland, Aug 2005.

[132] R. Marinescu and R. Dechter. Memory intensive branch-and-bound search for graphical models. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-06*, Boston, USA, 2006.

[133] R. Marinescu and R. Dechter. Best-first and/or search for graphical models. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-07*, Vancouver, Canada, 2007.

[134] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[135] R. Mateescu and R. Dechter. AND/OR cutset conditioning. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, Edinburgh, Scotland, Aug 2005.

[136] R. Mateescu and R. Dechter. Compiling Constraint Networks into AND/OR Multi-Valued Decision Diagrams (AOMDDs). In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06)*, Nantes, France, October 2006.

[137] A. Meisels and E. Kaplansky. Scheduling Agents Distributed Timetabling Problems (DisTTP). In *The Fourth International Conference on the Practice and Theory of Automated Timetabling (PATAT-2002)*, pages 166–180, Gent, Belgium, 2002.

[138] A. Meisels and I. Razgon. Distributed forward-checking with dynamic ordering. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2001, Seattle, USA*, 2001.

[139] A. Meisels and R. Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2003, Acapulco, Mexico*, 2003.

[140] P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. AAMAS*, 2003.

[141] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AI Journal*, 161:149–180, 2005.

[142] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7:95–132, 1974.

[143] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-93*, pages 40–45, Washington, DC, 1993. AAAI Press.

[144] R. Myerson and M. Satterthwaite. Efficient mechanisms for bilateral trading. *Journal of Economic Theory*, 28:265–281, 1983.

[145] M. V. Narumanchi and J. M. Vidal. Algorithms for distributed winner determination in combinatorial auctions. In *LNAI volume of AMEC/TADA*. Springer, 2006.

[146] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[147] R. Ostrovsky, S. Rajagopalan, and U. Vazirani. Simple and efficient leader election in the full information model. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 234–242, New York, NY, USA, 1994. ACM Press.

[148] J. Ostwald, V. Lesser, and S. Abdallah. Combinatorial Auction for Resource Allocation in a Distributed Sensor Network. *The 26th IEEE International Real-Time Systems Symposium, (RTSS'05)*, pages 266–274, December 2005.

[149] D. C. Parkes, J. R. Kalagnanam, and M. Eso. Achieving budget-balance with Vickrey-based payment schemes in exchanges. In *Proc. 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 1161–1168, 2001.

[150] D. C. Parkes and J. Shneidman. Distributed implementations of Vickrey-Clarke-Groves mechanisms. In *Proc. 3rd Int. Joint Conf. on Autonomous Agents and Multi Agent Systems*, pages 261–268, 2004.

[151] J. P. Pearce. USC DCOP repository, http://teamcore.usc.edu/dcop, 2005.

[152] P.E.Dunne. Extremal behaviour in multiagent contract negotiation. *Journal of Artificial Intelligence Research (JAIR)*, 23:41–78, 2005.

[153] P.E.Dunne, M.Wooldridge, and M.Laurence. The complexity of contract negotiation. *Artificial Intelligence Journal*, 164:23–46, 2005.

[154] A. Petcu. FRODO: A FRamework for Open and Distributed constraint Optimization. Technical Report No. 2006/001, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2006. `http://liawww.epfl.ch/frodo/`.

[155] A. Petcu and B. Faltings. Applying interchangeability techniques to the distributed breakout algorithm - poster. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI-03*, Acapulco, Mexico, 2003.

[156] A. Petcu and B. Faltings. A distributed, complete method for multi-agent constraint optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004)*, Toronto, Canada, Sep 2004.

[157] A. Petcu and B. Faltings. A value ordering heuristic for local search in distributed resource allocation. In *Recent Advances in Constraints, LNAI-3419*, pages 86–97. Springer Verlag, 2004.

[158] A. Petcu and B. Faltings. A-DPOP: Approximations in distributed optimization. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 802–806, Sitges, Spain, October 2005.

[159] A. Petcu and B. Faltings. A-DPOP: Approximations in distributed optimization. In *CP05 - workshop on Distributed and Speculative Constraint Processing, DSCP*, Sitges, Spain, October 2005.

[160] A. Petcu and B. Faltings. DPOP: A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, pages 266–271, Edinburgh, Scotland, Aug 2005.

[161] A. Petcu and B. Faltings. An efficient constraint optimization method for large multiagent systems. In *AAMAS05 - LSMAS workshop*, July 2005.

[162] A. Petcu and B. Faltings. Incentive compatible multiagent constraint optimization. In X. Deng and Y. Ye, editors, *LNCS 3828: WINE'05 - Workshop on Internet and Network Economics*, pages 708–717, Hong Kong, Dec 2005.

[163] A. Petcu and B. Faltings. LS-DPOP: A propagation/local search hybrid for distributed optimization. In *CP 2005- LSCS'05: Second International Workshop on Local Search Techniques in Constraint Satisfaction*, Sitges, Spain, October 2005.

[164] A. Petcu and B. Faltings. R-DPOP: Optimal solution stability in continuous time optimization. In *IJCAI05 - Distributed Constraint Reasoning workshop, DCR05*, Edinburgh, Scotland, August 2005.

[165] A. Petcu and B. Faltings. S-DPOP: Superstabilizing, fault-containing multiagent combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, pages 449–454, Pittsburgh, USA, July 2005.

[166] A. Petcu and B. Faltings. Distributed generator maintenance scheduling. In *Proceedings of the First International ICSC Symposium on ARTIFICIAL INTELLIGENCE IN ENERGY SYSTEMS AND POWER: AIESP'06*, Madeira, Portugal, Feb 2006.

[167] A. Petcu and B. Faltings. MB-DPOP: A memory-bounded extension of DPOP. In *In Proceedings of the Second International Workshop on Distributed Constraint Satisfaction Problems, ECAI'06*, Riva del Garda, Italy, August 2006.

[168] A. Petcu and B. Faltings. O-DPOP: An algorithm for Open/Distributed Constraint Optimization. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-06*, Boston, USA, July 2006.

[169] A. Petcu and B. Faltings. PC-DPOP: A partial centralization extension of DPOP. In *In Proceedings of the Second International Workshop on Distributed Constraint Satisfaction Problems, ECAI'06*, Riva del Garda, Italy, August 2006.

[170] A. Petcu, B. Faltings, and R. Dechter. DFS-based algorithms for (dynamic) distributed constraint optimization problems. *submitted to Artificial Intelligence Journal*, 2007.

[171] A. Petcu, B. Faltings, and D. Parkes. M-DPOP: Faithful Distributed Implementation of Efficient Social Choice Problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*, pages 1397–1404, Hakodate, Japan, May 2006.

[172] A. Petcu, B. Faltings, D. Parkes, and W. Xue. BB-M-DPOP: Budget-balance in social choice based on problem structure. Technical report id: Lia-report-2007-002, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), April 2007.

[173] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE,* April 2006.

[174] R. Porter, Y. Shoham, and M. Tennenholtz. Fair imposition. *Journal of Economic Theory*, 118:209–229, 2004.

[175] C.-F. Rey, A. Petcu, M. Schumacher, T. Leaute, and B. Faltings. Adaptive and distributed speed limitation in a simulation of the swiss highway network. Technical report lia-report-2007-004, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), 2007.

[176] G. Ringwelski and Y. Hamadi. Boosting distributed constraint satisfaction. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 549–562, Sitges, Spain, 2005.

[177] I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

[178] R. Rob. Asymptotic efficiency of the demand revealing mechanisms. *Journal of Economic Theory*, 28:207–220, 1982.

[179] K. Roberts. The characterization of implementable rules. In J.-J. Laffont, editor, *Aggregation and Revelation of Preferences*, pages 321–348. North-Holland, Amsterdam, 1979.

[180] E. Rollon and J. Larrosa. Depth-first mini-bucket elimination. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, Sitges, Spain, 2005. LNCS 2709.

[181] J. S. Rosenschein and G. Zlotkin. Designing conventions for automated negotiation. *AI Magazine*, 1994. Fall.

[182] M. H. Rothkopf, A. Pekec, and R. M. Harstad. Computationally manageable combinational auctions. *Manage. Sci.*, 44(8):1131–1147, 1998.

[183] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artif. Intell.*, 135(1-2):1–54, 2002.

[184] T. Sandholm and S. Suri. Bob: improved winner determination in combinatorial auctions and generalizations. *Artif. Intell.*, 145(1-2):33–58, 2003.

[185] T. Sandholm, S. Suri, A. Gilpin, and D. Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence, IJCAI-01*, pages 1102–1108, 2001.

[186] T. W. Sandholm. An implementation of the Contract Net Protocol based on marginal-cost calculations. In *Proc. 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 256–262, July 1993.

[187] T. W. Sandholm. Limitations of the Vickrey auction in computational multiagent systems. In *Second International Conference on Multiagent Systems (ICMAS-96)*, pages 299–306, 1996.

[188] T. W. Sandholm. Issues in computational Vickrey auctions. *International Journal of Electronic Commerce*, 4:107–129, 2000.

[189] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*, Montreal, Canada, 1995.

[190] P. Shenoy. Binary join trees for computing marginals in the Shenoy-Shafer architecture. *International Journal of Approximate Reasoning*, 17(2-3):239–263, 1997.

[191] J. Shneidman and D. C. Parkes. Rationality and self-interest in peer to peer networks. In *2nd Int. Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[192] J. Shneidman and D. C. Parkes. Specification faithfulness in networks with rational nodes. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing (PODC'04)*, St. John's, Canada, 2004.

[193] M. Silaghi. Generalized Dynamic Ordering for Asynchronous Backtracking on DisCSPs. In *AAMAS'06 - Distributed Constraint Reasoning workshop, DCR06*, Hakodate, Japan, May 2006.

[194] M. Silaghi and M. Yokoo. Nogood-based asynchronous distributed optimization (ADOPT-ng). In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*, Hakodate, Japan, May 2006.

[195] M. C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:25–53, 2005.

[196] M.-C. Silaghi, B. Faltings, and A. Petcu. Secure combinatorial optimization simulating DFS tree-based variable elimination. In *9th Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, Florida, USA, Jan 2006.

[197] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, Austin, Texas, 2000.

[198] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Distributed asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.

[199] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *2nd Asia-Pacific Conference on Intelligent Agent Technology (IAT'2001) World Scientific*, pages 54–63, 2001.

[200] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous consistency maintenance. In *2nd Asia-Pacific Conference on Intelligent Agent Technology (IAT'2001) World Scientific*, 2001.

[201] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report No. 364 364, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), may 2001.

[202] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *Proceedings of IAT*, 2004.

[203] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 561–562, Cambridge, Massachusetts, USA, 1996.

[204] K. Suzuki and M. Yokoo. Secure generalized vickrey auction using homomorphic encryption. In *Financial Cryptography*, pages 239–249, 2003.

[205] K. Sycara, S. F. Roth, N. Sadeh-Koniecpol, and M. S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1446–1461, December 1991.

[206] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[207] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 3(13), 1984.

[208] T. N. Tideman and G. Tullock. A new and superior process for making social choices. *Journal of Political Economy*, 84:1145–1159, 1976.

[209] T. N. Tideman and G. Tullock. Some limitations of demand revealing processes: Comment. *Public Choice*, 29:125–128, 1977.

[210] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[211] G. Verfaillie and N. Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, 2005.

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

[213] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[214] M. Walker. On the nonexistence of a dominant strategy mechanism for making optimal public decisions. *Econometrica*, 48:1521–1540, 1980.

[215] R. Walker. An enumerative technique for a class of combinatorial problems. Combinlatorial Analysis (Proceedings of Symposium in Applied Mathematics, Vol. X), Amer. Math. Soc., Providence, R.I.,, 1960.

[216] R. Wallace and E. C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Proceedings of CP98*, 1998.

[217] W. E. Walsh and M. P. Wellman. A market protocol for decentralized task allocation. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, page 325, Washington, DC, USA, 1998. IEEE Computer Society.

[218] M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[219] M. P. Wellman. Market-oriented programming: Some early lessons. In S. H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*, chapter 4, pages 74–95. World Scientific, 1996.

[220] M. P. Wellman, W. E. Walsh, P. R. Wurman, and J. K. MacKie-Mason. Auction protocols for decentralized scheduling. *Games and Economic Behavior*, 35:271–303, 2001.

[221] N. Wilson. Decision diagrams for the computation of semiring valuations. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, pages 331– 336, 2005.

[222] N. Yahfoufi and S. Dowaji. A self-stabilizing distributed branch-and-bound algorithm. In *Computers and Communications*, pages 246–252, 1996.

[223] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *CP '95: Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 88–102, London, UK, 1995. Springer-Verlag.

[224] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

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

[226] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In V. Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*. MIT Press, 1995.

[227] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 372–379, Paris, France, 1998.

[228] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[229] M. Yokoo, Y. Sakurai, and S. Matsubara. The effect of false-name bids in combinatorial auctions: New Fraud in Internet Auctions. *Games and Economic Behavior*, 46(1):174–188, 2004.

[230] M. Yokoo and K. Suzuki. Secure multi-agent dynamic programming based on homomorphic encryption and its application to combinatorial auctions. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 112–119, New York, NY, USA, 2002. ACM Press.

[231] M. Yokoo and K. Suzuki. Secure generalized vickrey auction using homomorphic encryption. *Financial Cryptography*, 2003.

[232] M. Yokoo and K. Suzuki. Secure generalized vickrey auction without third-party servers. In *Financial Cryptography, LNCS 3110*, pages 132–146, 2004.

[233] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *Proceedings of the Distributed Constraint Reasoning workshop at AAMAS 2002, Bologna*, 2002.

[234] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: reaching agreement without revealing private information. *Artif. Intell.*, 161(1-2):229–245, 2005.

[235] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artif. Intell.*, 161(1-2):55–87, 2005.

[236] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-02*, pages 352–357, Edmonton, Alberta, Canada, 2002.

[237] W. Zhang and L. Wittenburg. Distributed breakout algorithm for distributed constraint optimization problems - DBArelax. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, Melbourne, Australia, 2003.

[238] W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, pages 185–192, Melbourne, Australia, 2003.

[239] Y. Zhang and A. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. In L. N. Kanal, V. Kumar, H. Kitano, and C. B. Suttner, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Elsevier, Amsterdam, 1994.

[240] R. Zivan and A. Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, 2003.

[241] R. Zivan and A. Meisels. Concurrent Dynamic Backtracking for Distributed CSPs. In *Lecture Notes in Computer Science*, volume 3258, pages 782 – 787. Springer Verlag, Jan 2004.

[242] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints*, 11(2-3):179–197, 2006.

[243] R. Zivan and A. Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, April 2006.

[244] G. Zlotkin and J. S. Rosenschein. Mechanisms for automated negotiation in state oriented domains. *Journal of Artificial Intelligence Research*, 5:163–238, 1996.

List of Figures

List of Tables

List of Algorithms

Curriculum Vitae

ADRIAN PETCU

Ch. des Berges 10, 1022 Chavannes, Suisse *+41-21-6480666; apetcu@gmail.com*
Last update: November 19, 2007 http://liawww.epfl.ch/People/apetcu/

Expertise

- Artificial Intelligence / Combinatorial Optimization / Distributed Systems

Education

- **2002-2007: Swiss Federal Institute of Technology in Lausanne**—EPFL
  PhD in Artificial Intelligence*; advisor: Prof. Boi Faltings*
  - Thesis: A Class of Algorithms for Distributed Constraint Optimization
  - Nominated for the EPFL prize **"Best Thesis Of The Year"**.
- 1995-2000: "Politehnica" University Bucharest - ***MS in Computer Science***

Research

- Areas of Interest: Constraint Satisfaction/Optimization, Distributed/Multiagent Systems, Security/Privacy, Game Theory, Virtual Organizations, Cooperative Robotics
- Scientific Publications: over 30 reviewed papers in international conferences and workshops (see separate publications list, also available on my webpage)
- For an overview on some of my recent research work, please see
  `http://liawww.epfl.ch/Publications/Archive/Petcu2006d.pdf`

Honors and Awards

- PhD dissertation nominated for the EPFL prize **"Best Thesis Of The Year"**.
- **Scholarships** to participate in several conferences: CP'03, CP'04, AAMAS'06, ASAMAS'06
- Scholarship for top students during the 5 years in the Politehnica University in Bucharest
- Awarded a special prize for the graduation project

Patents

- Method to allocate inter-dependent resources by a set of participants

  US and PCT patent pending, 2007

  Boi Faltings, Thomas Leaute, Adrian Petcu

Academic Activities

- given a tutorial on Distributed Constraint Reasoning at IAT'07, Fremont, CA
- chair of the DCR'07 international workshop (in conjunction with IJCAI'07, India)
- co-organizer of the international DCSP'06 workshop (in conjunction with ECAI'06, Italy)

- co-organizer of the international CSCLP04 workshop: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming. The workshop included 22 papers and had 35 participants.
- co-editor of the Springer LNAI 3419 volume "CSCLP04: Recent Advances in Constraints"
- Reviewing: International Journals: AI Journal, Constraints Journal, Journal of AI Research (JAIR); International Conferences: CP 2004, IJCAI 2005, IJCAI 2007, AAAI 2007
- Involved in the European project AgentCities (http://www.agentcities.org): deployed a FIPA-compliant agent-based online banking service as one of the deliverables
- Supervised several students directly (Master thesis and semester projects)
- Teaching assistant for a PhD-level course (Distributed Information Processing: 2003, 2004, 2006) and one MS-level course (Intelligent Agents 2003)

Courses/seminars

- **Entrepreneurial course** "Venture Challenge" for creating a startup (by `VentureLab.ch`)
- **Language courses**: several series of courses of English, French, German, Italian
- **Communication courses** organized at EPFL: "Effective Communication", "Efficient Technical Presentations", "Straightforward English for professional writing"

Presentations

- **Invited talks**: Jan'06 LABOS group at EPFL, Switzerland; June'06 SAP Research Center in Karlsruhe, Germany; July'06, ECONCS group at Harvard University, Boston, USA
- **Oral presentations** in these international conferences: IJCAI'03, CSCLP'04, CP'04, IJCAI'05, AAAI'05, AAMAS'05, WINE'05, AAMAS'06, AAAI'06, ECAI'06, IJCAI'07
- **Poster presentations** in these international conferences: IJCAI'03, CP'03, CP'04, CP'05, AAMAS'06, AAAI'06, ASAMAS'06, IJCAI'07

Industrial Experience

- **2000-2002: eQuadriga AG Schweiz**—Project Manager, Software Designer, Network Expert
  - Project management for a 300000 EUR e-learning project; CRM, project life-cycle, planning, reporting, monitoring. Coordinating an off-shore team of 70 developers
  - High-level design of a multi-user eLearning platform (application and database level); implementation of small prototypes, integration of various technologies/modules
  - Design, deployment and maintenance of the computing infrastructure for a multi-national organization with several sites around the world, and around 100 employees (servers for mail/web/dns/proxy/samba, conference system, secure file transfers, multi-site concurrent development center, data-replication and backup)
- 1995-2000: Software-related internships and part-time jobs: Sysco SRL, Pepsi Cola Romania SA, General Turbo SA, Taxo Verlag Gmbh, Radiotel SA, Canad Systems Plus, Wizrom SA

Computer-related Skills

- Expert knowledge of Linux (Red Hat EL, Fedora, Ubuntu, SuSE), Windows, MacOS, Solaris

- **Computer Languages**: Java, C, C++, PHP, UNIX Shells, Perl, HTML, LaTeX, JSP
- **Tools, Systems**: Apache, BIND, CVS/RCS, Squid, Samba, SSHD, MySQL, NFS, Postfix

Languages

- **Romanian**: mother tongue; **English**: fluent (TOEFL:298/300, GRE verbal: 660/800); **French**: fluent (C2 level); **German**: good (B2+ level); **Italian**: good; **Spanish**: basic

Excellent communication skills:

- Fluent in 3 languages, good knowledge of other 3
- Taken a course on "Straightforward English for professional writing" organized at EPFL
- Taken a course on "Effective Communication" organized at EPFL
- Taken a course on "Efficient Technical Presentations" organized at EPFL
- Dozens of presentations/posters in international conferences and workshops
- Customer relations (including support) in an industrial environment for 1.5 years

Varia

- Romanian citizen, married, 2 children
- Swiss permanent residence permit type C
- Hobbies: Skiing, Aikido, Karate Shotokan, Target Shooting

References

- **Prof. Boi Faltings**, head of the Artificial Intelligence Lab, **EPFL**
  Phone: (+41 21) 693-2735, Fax (+41 21) 693-5225, Email: boi.faltings@epfl.ch
- **Prof. David C. Parkes**, Division of Engineering and Applied Science, **Harvard University**
  Phone: (617) 384-8130, Fax: (314) 248-7899, Email: parkes@eecs.harvard.edu
- **Prof. Makoto Yokoo**, Department of Intelligent Systems, **Kyushu University**
  Phone: (+81)-92-642-4065, Fax: (+81)-92-632-5204, Email: yokoo@is.kyushu-u.ac.jp
- **Prof. Marius Silaghi, Florida Institute of Technology**
  Phone: (+1-321)-674-7493, Fax: (+1-321) 674-7046, Email: Marius.Silaghi:@cs.fit.edu
- **Erwin Selg**, CTO at **GFT Technologies AG**, Germany
  Phone: +49 711 6242 436, Fax: +49 711 6242 101, Email: erwin.selg@gft.com
- **Eugen Serbanescu**, Project Manager, **Nortel Networks** Romania
  Phone: +40-21 327 22 85, Fax: +40-21 327 22 89, Email: eserbanescu@nortel.com