UNIVERSITÀ DEGLI STUDI DI UDINE
DIPARTIMENTO DI MATEMATICA E INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS

Exploiting the Structure of Distributed Constraint Optimization Problems

CANDIDATE:
Ferdinando Fioretto

SUPERVISORS:
Prof. Agostino Dovier
Prof. Enrico Pontelli

Academic Year 2015–2016

Author's e-mail: ffiorett@cs.nmsu.edu

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Department of Computer Science
New Mexico State University
Box 30001, MSC CS
Las Cruces, NM, 88003 - U.S.A.

Abstract

Distributed Constraint Optimization Problems (DCOPs) have emerged as one of the prominent multi-agent architectures to govern the agents' autonomous behavior in a Multi-Agent System (MAS), where several agents coordinate with each other to optimize a global cost function. They represent a powerful approach to the description and resolution of many practical problems, and serve several applications such as distributed scheduling, coordination of unmanned air vehicles, smart grid electric networks, and sensor networks. Typical real world applications are characterized by complex dynamics and interactions among a large number of entities, which translate into hard combinatorial problems, posing significant challenges from a computational point of view.

The adoption of DCOPs on large instances of problems faces two main limitations: *(1) Modeling limitations*, as current resolution methods detach the model from the resolution process, imposing limiting assumptions on the capabilities of an agent (e.g., that it controls a single variable of the problem, and that it operates solely on the resolution of a global problem, ignoring the presence of private objectives); and *(2) Solving capabilities*, as the inability of current approaches to capitalize on the presence of structural information which may allow incoherent/unnecessary data to reticulate among the agents as well as to exploit latent structure of the agent's local problems, and/or of the problem of interest.

The objective of the proposed dissertation is to address such limitations, studying how to adapt and integrate insights gained from centralized solving techniques, and from General Purpose Graphic Processing Units (GPGPUs) parallel architectures, in order to design practical algorithms to efficiently solve large, complex, DCOPs, enabling their use for the resolution of real-world problems. To do so, we hypothesize that one can exploit the latent structure of DCOPs in both problem modeling and problem resolution phases.

Acknowledgments

I am indebted to my advisors Enrico Pontelli and Agostino Dovier for their guidance and encouragement in my research and beyond, and for making this dissertation possible.

Contents

List of Figures

List of Tables

1 Introduction

The field of *Multi-Agent System (MAS)* is an active area of research within Artificial Intelligence, with an increasingly important impact in industrial and other real-world applications. Within a MAS, autonomous agents interact to pursue personal interests and/or to achieve common objectives. *Distributed Constraint Optimization Problems (DCOPs)* [81, 96, 124] are problems where several agents coordinate with each other to optimize a global cost function, and have emerged as one of the prominent agent architectures to govern the cooperative agents' autonomous behavior. DCOPs are characterized by four components: (1) *agents*, (2) *variables*, (3) *domains*, and (4) *constraints*. An *agent* can be defined as an entity (or computer program) that behaves autonomously within an arbitrary system in the pursuit of some goals. Each DCOP agent controls a set of variables, which model the entities of the problem. Each variable can take values from a finite set of elements, which defines its domain. In a DCOP, each variable is controlled by an agent, and thus the decision of the value assignment for a variable is made exclusively by the agent controlling it. In a DCOP, relations are expressed among subsets of the problem variables, and specify the utilities gained by the agents when they coordinate the assignments for such variables. Thus, DCOP agents need to coordinate their value assignments, in a decentralized manner, to optimize their objective functions. The common mean to execute such distributed coordination process is via agents' communication, conducted through the exchange of messages. DCOPs focus on attaining a global optimum given the interaction graph of a collection of agents. This approach is flexible and can effectively model a wide range of practical problems, such as scheduling problems [73, 132], resource allocation [31, 131], and power network management problems [62].

The focus of the work presented in this dissertation is the investigation of solution approaches to efficiently solve DCOPs. To this end, we focus on exploiting latent structure, exposed by the constraints found within general distributed optimization problems, or exposed by the specific application of interest, and on exploiting the use of *General Purpose Graphic Processing Units (GPGPUs)* to enhance DCOP algorithms solving efficiency.

1.1 Research Objective

In recent years, it has been observed a transition within the Distributed Constraint Optimization community, from theory and algorithm's development to practical applications of DCOP algorithms. Typical real world applications are characterized by complex dynamics and interactions among a large number of entities, which translate into hard combinatorial problems, posing significant challenges from a computational point of view. In this dissertation we identified two limitations faced by the adoption of DCOP on large problems: *(1) Modeling limitations*, as current resolution methods detach the model from the resolution process, imposing limiting assumptions on the capabilities of an agent (e.g., that it controls a single variable of the problem); and *(2) Solving capabilities*, as the inability of current approaches to capitalize on the presence of structural information which may allow incoherent/unnecessary data to reticulate among the agents as well as to exploit structure of the agent's local problems.

The research objective of the proposed dissertation is to address such limitations. To do so, we hypothesize that:

1. One can exploit the information encoded in the DCOP model through the use of centralized solutions.

2. One can adapt centralized reasoning techniques to exploit the structure of DCOPs during problem solving.

3. One can exploit highly parallel computational models to enhance current DCOP solution techniques.

Thus, our focus is to study how to adapt and integrate insights gained from centralized solving techniques and from General Purpose Graphic Processing Units (GPGPUs) parallel architectures in order to enhance DCOP performance and scalability, enabling their use for the resolution of real-world complex problems.

The study described in this dissertation benefits both the theory and the practice of Multi-Agent Systems and High Performance Computing, and helps application domains such as distributed scheduling and allocation, smart grid electricity networks, logistics and sensor networks, where researcher often encounter the need of distributed or parallel solutions to handle large-scale complex optimization problems.

1.2 Contributions

We describe below the main contributions of this dissertation.

1.2.1 Exploiting Structure from Problem Modeling

Modeling many real-world problems as DCOPs often requires each agent to control a large number of variables. However, most DCOP resolution approaches assume that each agent controls exclusively a single variable of the problem. As such, researchers have proposed a number of pre-processing techniques to reformulate DCOPs with multi-variable agents into DCOPs with single-variable agents [125]. Unfortunately, these techniques do not scale with the size of the problem due to their inefficient communication requirements. Therefore, we proposed a DCOP *Multiple-Variable Agents* (MVA) problem decomposition [37] that defines a clear separation between the distributed DCOP resolution and the centralized agent sub-problem resolution. This separation exploits co-locality of agent's variables, allowing the adoption of efficient centralized techniques to solve agent sub-problems, while preserving agent's privacy. Agents coordination is achieved employing a global DCOP algorithm [36, 37]. Using the MVA problem decomposition, allows us to significantly reduce the time of the DCOP resolution process. In addition, the knowledge acquired from the DCOP model allows us to further reduce the algorithms communication requirements, when compared to existing pre-processing techniques—which ignore the structural information dictated by the model.

These results validate our hypothesis that one can exploit the information encoded in the DCOP model through the use of centralized solutions.

1.2.2 Exploiting Structure during Problem Solving

A number of multi-agent systems require agents to run on battery-powered devices and communicate over wireless networks. This imposes constraints on the number and size of individual messages exchanged among agents. *Inference-based* DCOP algorithms, can be effective in solving such problems. They use techniques from dynamic programming to propagate aggregate information among agents, and while their requirements on the number of messages is linear in the number of agents, their messages have a size that is exponential in the size of the treewidth, which can be up to the number of agents -1. Several works from the DCOP community recognize the use of hard constraints to reduce the size of the search space and/or reduce the message size. However, they are limited in exploiting relational information expressed in form of tables and/or associated to the form of domain consistency. In response of such limitation we introduce a type of consistency, called *Branch Consistency* [35], that applies to paths in pseudo-trees. The effect of enforcing Branch Consistency is the ability to actively exploit hard constraints (either explicitly provided in the problem specification or implicitly described in constraints cost tables) to prune the search space and to reduce the size of the messages exchanged among agents. Such form of consistency enforces a more effective pruning than those based on domain-consistency, guaranteeing optimality, and leading enhanced efficiency and scalability.

However, solving DCOPs optimally is NP-hard, therefore for large problems, incomplete DCOP algorithms are desirable. Several incomplete approaches have been proposed in the literature, yet, current incomplete DCOP algorithms have combinations of the following limitations:

i. They find local minima without quality guarantees;

ii. They provide loose quality assessment, such us those in the class of k-optimality [91];

iii. They do not exploit problem structures, such as those induced by the problem domain, or by hard constraints [84].

Therefore, building on strategies from the centralized constraint reasoning community, we introduce the *Distributed Large Neighborhood Search* (D- LNS) framework [32]. D-LNS solves DCOPs by building on the strengths of centralized *Large Neighboring Search strategy (LNS)* [55], a centralized meta-heuristic that iteratively explores complex neighborhoods of the search space to find better candidate solutions. D-LNS is a local search framework for DCOPs, which has several qualities:

- It provides quality guarantees by refining both upper and lower bounds of the solution found during the iterative process;
- It is anytime (i.e., it is able to return a valid solution to the problem even if it is interrupted at any time before it ends); and

- It inherently uses insights from the *CP* techniques to take advantage on the presence of hard constraints.

In addition we introduce two novel distributed search algorithms, built within the D-LNS framework, characterized by the ability to exploit problem structure, low network usage, and low computational complexity per agent. Compared to other incomplete algorithms, our D-LNS converges faster to better solutions, provides tighter solution quality bounds, is more scalable, and it can exploit explicitly domain dependent knowledge, to further enhance runtime and solution quality of the problem being solved.

These results validate our hypothesis that centralized reasoning can be adapted to exploit the structure of DCOPs during problem solving.

1.2.3 Exploiting the use of GPGPUs

Typical Distributed Constraint Optimization problems are characterized by complex dynamics and interactions among a large number of agents, which translate into hard combinatorial problems, posing significant challenges from a computational point of view. To deal with such computational burden, in addition to the techniques discussed above, we exploit a novel class of massively parallel platforms that are based on the *Single Instruction Multiple Thread* (SIMT) paradigm, and widely used in modern GPGPUs. GPGPUs are multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy supporting *general purpose* (i.e., non-graphical) processing. The wide availability of GPGPUs, and their contained costs, stimulated interest across several research communities.

The structure exploited by Dynamic Programming (DP)-based approaches in constructing solutions makes it suitable to exploit the SIMT paradigm. Thus, we propose a DP-based algorithm that exploits parallel computation using GPGPUs to solve DCOPs [34]. Our proposal employs GPGPU hardware to speed up the inference process of DP-based methods. Our results show significant improvements in performance and scalability over other state-of-the-art DP-based solutions.

The explicit separation between the DCOP resolution process and the centralized agent problem, enabled by our MVA DCOP decomposition, capacitate agents to solve their local problem through a variety of techniques. Motivated by the high complexity of the agent local problem, we propose the use of hierarchical parallel models, where each agent can:

i. Solve its local problem independently from those of other agents, and

ii. Parallelize the computations within its own local problem.

Such model builds on top of algorithm-specific characteristics, and may substantially reduces the runtime for several DCOP algorithms classes. Thus, we suggest to solve independent local problems, in parallel, harnessing the multitude of computational units offered by GPGPUs, which leads to significant improvements in the runtime of the algorithm resolution [33, 37] .

These results validate our hypothesis that one can exploit highly parallel computational models to enhance current DCOP solution techniques.

1.3 Dissertation Organization

This dissertation is organized as follows: The next chapter (Chapter 2) provides an overview of the distributed constraint optimization model and algorithms, and of GPGPUs. Therein, we review the notions of centralized constraint optimization problems and constraint solving, and discuss some general techniques that are typically adopted to solve constraint problems (e.g., constraint propagation and search). We hence discuss the distributed constraint optimization model, the typical representation and coordination schema adopted during the resolution process and we review a number of complete and incomplete DCOP algorithms. We further discuss the DCOP model extensions able to deal with dynamic and uncertain events. Finally, we provide an overview of the graphical processing units, and review the details of such architecture and of its different memory levels. Chapter 3 introduces a *Multiple Variable Agents* (MVA) decomposition technique for DCOPs to exploit the information encoded in the DCOP model through the use of centralized solutions. Chapter 4 introduces *Branch Consistency* (BrC), and *Distributed Large Neighboring Search* (D-LNS), two DCOP solving strategies which adapt centralized reasoning techniques to enhance the efficiency of DCOP resolution by exploiting the structure of DCOPs in orthogonal ways. Chapter 5 proposes the design and implementation of inference-based and sampling-based algorithms which exploits modern massively parallel architectures, such as those found in modern General Purpose Graphical Processing Units (GPGPUs), to speed up the resolution of DCOPs. Finally, we conclude the dissertation in Chapter 6, which summarizes the main results presented in the previous chapters and identifies possible directions for future work.

To facilitate the reading of this dissertation, we have provided in the Appendix A a summary of the most commonly used notations.

2
Background

This chapter aims at providing an overview of Distributed Constraint Optimization (section 2.1), addressing the resolution methods that are central for the development of this dissertation. In section 2.2, we propose a classification of DCOP models from a Multi-Agent Systems perspective, providing an overview of some recent DCOP extensions, which enrich the original DCOP model expressiveness and applicability. Finally, we provide some background on General Purpose Graphical Processing Units, in section 2.3, which are used within several approaches developed throughout this work, to speed up the resolution approach of various DCOP algorithms.

2.1 Overview of Distributed Constraint Optimization

In this section, we provide an overview of Constraint Programming, which forms the foundation of Distributed Constraint Optimization. We thus describe Distributed Constraint Optimization Problems their representation and coordination models, resolution approaches, and relevant uses.

2.1.1 Constraint Programming

Constraint Programming (CP) is a declarative programming methodology. Over the years, CP has become a paradigm of choice to address hard search problems, drawing and integrating insights from diverse domains, including Artificial Intelligence and Operations Research [107]. The basic idea in this programming paradigm relies on the use of relations that should hold among entities of the problem—this notion is referred to as *Constraint Satisfaction Problem* (CSP).

Constraint Satisfaction and Optimization

A CSP is a triple $P = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, where:

- $\mathbf{X} = \{x_1, \dots, x_n\}$ is a finite set of variables.
- $\mathbf{D} = \{D_1, \dots, D_n\}$ is a set of finite domains for the variables in $\mathbf{X}$, with D_i being the set of possible values for the variable x_i.
- $\mathbf{C}$ is a finite set of constraints over subsets of $\mathbf{X}$, where a constraint c_i defined on the m variables $x_{i_1}, \dots, x_{i_m}$, is a relation $c_i \subseteq \times_{j=1}^{m} D_{i_j}$. The set of variables $\mathbf{x}^i = \{x_{i_1}, \dots, x_{i_m}\}$ is referred to as

the *scope* of c_i. If $m = 1$, c_i is called *unary* constraint; if $m = 2$, it is called *binary* constraint. For all other $m > 2$, the constraint is referred to as *global* constraint.

A *solution* is a value assignment for a subset of variables from $\mathbf{X}$ that is consistent with their respective domains; i.e., it is a partial function $\theta : \mathbf{X} \rightarrow \bigcup_{i=1}^{n} D_i$ such that, for each $x_j \in \mathbf{X}$, if $\theta(x_j)$ is defined, then $\theta(x_j) \in D_j$. A solution is *complete* if it assigns a value to each variable in $\mathbf{X}$. We will use the notation σ to denote a complete solution, and, for a set of variables $\mathbf{V} = \{x_{i_1}, \dots, x_{i_h}\} \subseteq \mathbf{X}$, $\sigma_{\mathbf{V}} = \langle \sigma(x_{i_1}), \dots, \sigma(x_{i_h}) \rangle$, where $i_1 < \cdots < i_h$, denoting the projection of the values in σ associated to the variables in $\mathbf{V}$. The goal in a CSP is to find a complete solution σ such that for each $c_i \in \mathbf{C}$, $\sigma_{\mathbf{x}^i} \in c_i$, that is, one that satisfies all the problem constraints.

A CSP may be associated to an optimization function g. Informally, the goal of solving such CSPs does not simply rely on finding some solution, but on finding an optimal one, according to some optimization criteria g. We refer to such extended CSPs, to as *Constraint Optimization Problems* (COPs). A COP is a pair (P, g), where $P = (\mathbf{X}, \mathbf{D}, \mathbf{C})$ is a CSP, and $g : \bigtimes_{i=1}^{n} D_i \rightarrow \mathbb{R}$ is an *optimization function*. Solving a COP (P, g) means finding a solution s to P such that $g(s)$ is maximal (or minimal, depending on the optimization criteria) among all solutions of P.

As an example consider the classical *knapsack problem*, where we are given a set of items, each with a weight and a value. The goal is that of determining the quantity of each item to include in a collection so that the total weight does not exceed a given limit and the total value is as large as possible [57]. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W, the problem can be expressed as follows:

$$\begin{aligned} \textit{maximize:} \ & \sum_{i=1}^{n} v_i x_i \\ \textit{subject to:} \ & \sum_{i=1}^{n} w_i x_i \leq W, \text{and } x_i \in \{0, N_i\}. \end{aligned}$$

where x_i represents the number of instances of item i to include in the knapsack, and N_i is the maximum number of of copies of x_i that can be considered in the final collection.

Constraint Propagation

Conventional methods adopted to solve a constrained problem rely on a form of constraint reasoning to transform the original CSP in a new simpler, yet equivalent one—that is, one that preserves all the solutions. The idea of simplicity of a CSP typically refers to narrow variables' domains. *Constraint propagation* is a technique used to achieve this goal. It embeds any form of reasoning to explicitly preclude the use of those variables' values that would prevent a constraint to be satisfied. This process is performed by repeatedly narrowing domains, and/or constraints, while maintaining the CSP equivalence. To do so, the process of propagation needs to guarantee some form of *local consistency*, which ensures that some subset of variables satisfies the notion of the constraints in which they are involved.

We now review some popular form of consistency: *node consistency, arc consistency* and *path consistency*.

Node consistency is the simplest notion of local consistency. A CSP (X, D, C) is said to be node consistent if for every variable $x_i \in X$, D_i is consistent with every unary constraint c_i on x_i, i.e.,

$\forall a \in D_i,\ a \in c_i$; Node consistency can be achieved in $\mathcal{O}(nd)$ time [71], where $d = \max_i |D_i|$ is the maximum size among all variables' domains, by removing the inconsistent values from the domain of each variables with unary constraints.

Arc consistency applies to binary constraints. A binary constraint c_{ij} on variables x_i and x_j is *arc consistent* if and only if for every value $a \in D_i$ there exists a value $b \in D_j$ such that $(a, b) \in c_{ij}$, and for every value $b \in D_j$ there is a value $a \in D_i$ such that $(a, b) \in c_{ij}$. Similarly to node consistency, a CSP is *arc consistent* if all its binary constraints are arc consistent. Typical algorithms to achieve arc consistency rely on iterative processes handling one constraint at a time, and removing inconsistent values of the variables involved in the scope of the constraint [70, 6]. Such process repeats until a fixed point is reached, that is, when either no further pruning is possible or when some domain results empty.

Path consistency considers pairs of variables, in contrast to arc consistency which considers single variables. A pair of values $(r, c) \in D_i \times D_j$ of two variables x_i, x_j is path consistent if and only if for any sequence of variables $(x_i = x_{k_1}, \ldots, x_{k_m} = x_j)$, such that $c_{k_p k_q} \in \mathbf{C}$, where $p \leq q \leq p+1$, there exists a tuple of values $(r = v_{k_1}, \ldots, v_{k_m} = c)$ such that $v_{k_q} \in D_{k_q}$ and $(v_{k_p}, v_{k_q}) \in c_{k_p k_q}$, for each $1 \leq q \leq m$ and $p \leq q \leq p+1$. A CSP is *path consistent* if and only if for any pair of variables (x_i, x_j), with $i \leq j$, any locally consistent pair of values on (x_i, x_j) is path consistent. Differently from node and arc consistency, where the form of constraint propagation enforcing them works by removing inconsistent values from the variables domains, propagation in path consistency works by removing inconsistent assignments from a constraint [82].

The interested reader can refer to [7, 2] and references therein, for an in-depth analysis on local consistencies and constraint propagation.

Search

The common resolution process of a CSPs/COPs is typically expressed by a search process, where the values for the variables of the problem are generated in some (possibly) systematic order. We now briefly review major complete and incomplete approaches.

Complete Search: The resolution process of CSP/COP can be typically described by the iteration of two phases: *constraint propagation*, and *labeling*. The former, as described above, aims at reducing the domain of the variables not yet assigned, while the latter enforces a value choice for some variable of the problem. Thus, solving a CSP/COP can be expressed as the process of exploring a search tree (referred to as *prop-labeling* tree) where constraint propagation phases are interleaved with non-deterministic branching phases used to explore different value assignments to variables [2]. Each node of the prop-labeling tree represents a possible value assignment for a variable, the arcs connecting nodes express the effect of propagating constraints, and a solution is described by a complete path from the root node to a leaf node.

Such type of search space exploration is typically referred to *backtracking search* [85]. In backtracking search, a solution is constructed by extending a partial assignments to the problem's variables, and a backtrack to previous assignments is enforced as soon as the current assignment causes the violation of some constraint, or when no more values assignments can be attempted for a given variable. Several heuristics can be chosen to select the next node to expand (i.e., the next variable to label), or the next value to assign to a given variable. We refer the interested readers to [7, 2] and references therein for a detailed description on search strategies adopted in the resolution of a CSP/COP.

Local Search: By systematically exploring each path of the search tree, complete search strategies generate all possible solutions of a given problem. Since solving optimally a CSP/COP is NP-Complete [26] incomplete solution approaches are often necessary to solve large interesting problems. Local search (LS) methods [1, 92, 80] attempt to improve a current solution by iteratively generating new candidate solutions. They rely on the intuition that it is possible to navigate different regions of the search space by modifying some ”subsequences” (i.e., assignments for a subset of the problem variables) of the current solution, and possibly generating new better candidate solutions. The set of subsequences that can be modified is referred to as *neighborhood*.

We now review a widely adopted local search technique to tackle large constraint optimization problems: the *Large Neighborhood Search* (LNS) [109, 55]. In LNS an initial solution is iteratively improved by repeatedly *destroying* it and *repairing* it. Destroying a solution means selecting a subset of variables whose current values will be discarded. The set of such variables is referred to as *large neighborhood (LN)*. Repairing a solution means finding a new value assignment for the destroyed variables, given that the other non-destroyed variables maintain their values from the previous iteration. The peculiarity of LNS, compared to other local search techniques, is the (larger) size of the neighborhood to explore at each step. This method relies on the intuition that searching over a larger neighborhood allows the process to escape local optima and find better candidate solutions.

2.1.2 Distributed Constraint Optimization Problems

When the elements of a COP are distributed among a set of autonomous agents, we refer to it as Distributed Constraint Optimization Problem (DCOP).

Formally, a DCOP is described by a tuple $P = (\mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha)$, where $\mathbf{X}$ and $\mathbf{D}$ are the set of variables and their domains defined as in classical COPs, $\mathbf{F} = \{f_1, \dots, f_k\}$ is a finite set of *function*, with $f_i : \times_{x_j \in \mathbf{x}^i} D_j \rightarrow \mathbb{R}^+ \cup \{\bot\}$, where $\bot$ is a special element used to denote that a given combination of values for the variables in $\mathbf{x}^i$ is not allowed[1], $\mathbf{A} = \{a_1, \dots, a_p\}$ is a finite set of autonomous agents, and $\alpha : \mathbf{X} \rightarrow \mathbf{A}$ is a surjective function, from variables to agents, which assigns the control of each variable $x \in \mathbf{X}$ to an agent $\alpha(x)$.

Each function f_i represents a factor in a *global objective function*, $g(\mathbf{X}) = \sum_{i=1}^{k} f_i(\mathbf{x}^i)$. In the DCOP literature, the weighted constraints f_i are also called *constraints*, *cost functions*, *utility functions*, or *reward functions*. With a slight abuse of notation, we will denote with $\alpha(f_i)$ the set of agents whose variables are involved in the scope of f_i, i.e., $\alpha(f_i) = \{\alpha(x) \mid x \in \mathbf{x}^i\}$. When clear from the context, we will write $f_{\mathbf{x}^i}$ to refer to the function $f_i \in \mathbf{F}$ whose scope is $\mathbf{x}^i$. For instance, we will denote with f_{12} the binary function involving variables x_1 and x_2.

The goal in a DCOP is to find a complete solution that maximizes the total problem reward expressed by its reward functions:

$$\sigma^* = \operatorname*{argmax}_{\sigma \in \mathbf{\Sigma}} g(\sigma) = \operatorname*{argmax}_{\sigma \in \mathbf{\Sigma}} \sum_{f_i \in \mathbf{F}} f_i(\sigma_{\mathbf{x}^i}), \tag{2.1}$$

where Σ is the *state space*, defined as the set of all possible complete solutions. Analogously, for minimization problems, the argmax of the above expression is substituted with the argmin. Typically, the objective functions values of a maximization problem are referred to as *utilities*, while in a minimization problem they are referred to as *costs*.

[1] We assume sets of variables to be sorted according to a fixed order of $\mathbf{X}$.

Let us also introduce the following notations. Given an agent a_i, we denote with $N_{a_i} = \{a'_i \in \mathbf{A} \,|\, a_i \neq a'_i,\ \exists f_j \in \mathbf{F},\ x_r, x_s \in \mathbf{x}^j.\ \alpha(x_r) = a_i \wedge \alpha(x_s) = a'_i\}$ the set of its *neighboring agents*. A constraint f_i is said to be *hard* if $\forall \sigma \in \Sigma$ we have that $f_i(\sigma_{\mathbf{x}^i}) \subseteq \{0, \perp\}$. Otherwise, the constraint is said to be *soft*.

We refer to a *DCOP algorithm* as a distributed algorithm for the resolution of a DCOP.

2.1.3 Representation and Coordination

Representation in DCOPs plays a fundamental role, both from an agent coordination perspective and from an algorithmic perspective. We discuss here the most predominant representations adopted in various DCOP algorithms. Let us start by describing some widely adopted assumptions regarding agent knowledge and coordination, which will apply throughout this document, unless otherwise stated:

1. A variable and its domain are known exclusively to the agent controlling it and its neighboring agents.
2. Each agent knows the reward values of the constraints involving at least one of its local variables. No other agent has knowledge about such constraints.
3. Each agent knows exclusively (and it may communicate with) its own neighboring agents.

Constraint Graph Given a DCOP P, $G_P = (\mathbf{X}, E_C)$ is the *constraint graph* of P, where an undirected edge $\{x, y\} \in E_C$ exists if and only if there exists $f_j \in \mathbf{F}$ such that $\{x, y\} \subseteq \mathbf{x}^j$. A constraint graph is a standard way to visualize a DCOP. It underlines the agents' locality of interactions and therefore it is commonly adopted by DCOP resolution algorithms.

Given an ordering o on $\mathbf{X}$, we say that a variable x_i has a higher *priority* with respect to a variable x_j if x_i appears before x_j in o. Given a constraint graph G_P and an ordering o on its nodes, the *induced graph* G^*_P on o, is the graph obtained by connecting nodes, processed in increasing order of priority, to all their higher-priority neighbors. For a given node, the number of higher-priority neighbors is referred to as its *width*. The *induced width* w^*_o of G_P is the maximum width over all the nodes of G^*_P on ordering o.

Figure 2.1(a) shows an example constraint graph of a DCOP with four agents a_1 through a_4, each controlling one variable with domain $\{0,1\}$. There are two constraint: a ternary constraint, f_{123} with scope $\mathbf{x}^{123} = \{x_1, x_2, x_3\}$ and represented by a clique among x_1, x_2 and x_3, and a binary constraint f_{24} with scope $\mathbf{x}^{24} = \{x_2, x_4\}$.

Pseudo-Tree A number of DCOP algorithms require a partial ordering among the agents. In particular, when such order is derived from a depth-first search exploration, the resulting structure is known as *(DFS) pseudo-tree*. A pseudo-tree arrangement for a DCOP P is a subgraph $T_P = \langle \mathbf{X}, E_T \rangle$ of G_P such that T_P is a spanning tree of G_P—i.e., a connected subgraph of G_P containing all the nodes and being a rooted tree—with the following additional condition: for each $x, y \in \mathbf{X}$, if $\{x, y\} \subseteq \mathbf{x}^i$ for some $f_i \in \mathbf{F}$, then x, y appear in the same branch of T_P (i.e., x is an ancestor of y in T_P or vice versa). Edges of G_P that are *in* (respectively *out* of) E_T are called *tree edges* (respectively *backedges*). The tree edges connect parent-child nodes, while backedges connect a node with its *pseudo-parents* and its *pseudo-children*. We use C_{a_i}, PC_{a_i}, P_{a_i}, PP_{a_i}, to denote the set of children, pseudo-children, parent and pseudo-parents of the agent a_i.

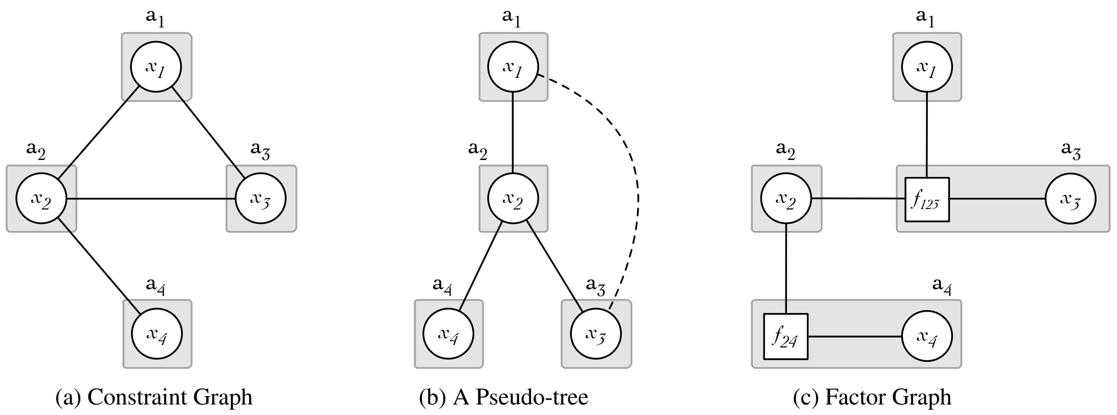


Figure 2.1: DCOP representations: An example constraint graph of a DCOP (a), one of its possible pseduo-trees (b), and its factor graph (c).

Both constraint graph and pseudo-tree representations cannot deal explicitly with n-ary constraints (functions whose scope has more than two variables). A typical artifact to deal with n-ary constraints in a pseudo-tree representation is to introduce a virtual variable which monitors the value assignments for all the variables in the scope of the constraint, and generates the reward values [10]—the role of the virtual variables can be delegated to one of the variables participating in the constraint [93, 76].

Figure 2.1(b) shows one possible pseudo-tree of the example DCOP in Figure 2.1(a), where $C_{a_1} = \{x_2\}$, $PC_{a_1} = \{x_3\}$, $P_{a_4} = \{x_2\}$, and $PP_{a_3} = \{x_1\}$. The solid lines are tree edges and dotted lines are backedges.

Factor Graph Another way to represent DCOPs is through a *factor graph* [61]. A *factor graph* is a bipartite graph used to represent the factorization of a function. In particular, given the global objective function g, the corresponding factor graph $F_P = \langle \mathbf{X}, \mathbf{F}, E_F \rangle$ is composed of variable nodes $x_i \in \mathbf{X}$, factor nodes $f_j \in \mathbf{F}$ and edges E_F such that there is an undirected edge between factor node f_j and variable node x_i if $x_i \in \mathbf{x}^j$.

Factor graphs can handle n-ary constraints explicitly. To do so, they use a similar method as that adopted within pseudo-trees with n-ary constraints: they delegate the control of a factor node to one of the agents controlling a variable in the scope of the constraint. From an algorithmic perspective, the algorithms designed over factor graphs can directly handle n-ary constraints, while algorithms designed over pseudo-trees require changes in the algorithm design so to delegate the control of the n-ary constraints to some particular entity.

Figure 2.1(c) shows the factor graph of the example DCOP in Figure 2.1(a), where each agent a_i controls its variable x_i and, in addition, a_3 controls the constraint f_{123} and a_4 controls f_{24}.

To facilitate the reading of this dissertation, we have provided in Table A.1, a summary of the most commonly used notations.

2.1.4 DCOP Algorithms

DCOP algorithms can be classified as being either *complete* or *incomplete*, based on whether they can guarantee the optimal solution or they trade optimality for smaller use of resources, producing approximated solutions. In addition, each of these classes can be categorized into several groups, such as: **(1)** *partially* or *fully decentralized*, depending on the degree of locality exploited by the algorithms; and **(2)** *synchronous* or *asynchronous*, based on the way local information is updated. Finally, the resolution process adopted by each algorithm can be classified in three categories [120]:

- *Search-based methods*, which are based on the use of search techniques to explore the space of possible solutions. These techniques are often derived from corresponding search techniques developed for centralized AI search problems, such as best-first search and depth-first search.
- *Inference-based methods*, which are inspired from dynamic programming and belief propagation techniques. These techniques allow agents to exploit the structure of the constraint graph to aggregate rewards from their neighbors, effectively reducing the problem size at each step of the algorithm.
- *Sampling-based methods*, which are incomplete approaches that sample the search space to approximate a function (usually a probability distribution) as a product of statistical inference.

Figure 2.2 illustrates a taxonomy of classical DCOP algorithms. In the following subsections, we describe the criteria adopted to evaluate the DCOP algorithms performance, and describe some representative complete and incomplete algorithms of each of the classes introduced above. For a detailed description of the DCOP algorithms we refer the interested readers to the original articles that introduce each algorithm.

Evaluation Criteria

In *centralized* optimization the performance of an algorithm is typically evaluated measuring the quality of the solution returned by the algorithm and its runtime. In addition to the solution quality, due to the distributed nature of the DCOPs, DCOP algorithms are generally examined employing two evaluation criteria: *execution time*, and *network performance*.

In the literature the distributed execution of the DCOP algorithms is often simulated on a single machine, thus a *simulated* execution time metric is often adopted. There are two widely adopted simulated execution time metrics: the *simulated runtime* [111], and the *non-concurrent constraint checks* (NCCCs) [66], and are defined as follows:

- *Simulated runtime* measures both processing and communication time of the DCOP algorithm. Every agent a_i maintains an internal clock t_i, which is initialized to 0. When an agent performs some operation it measures the runtime elapsed and adds it to t_i. When an agent a_i sends a message to some agent a_j it also forwards its current timer t_i. When a_i receives a message from a_j it updates its current timer $t_i = \max\{t_i, t_j + D\}$, where D is a delay time used to simulate the communication time. The simulated runtime of the algorithm is the largest timer held by any agent in $\mathbf{A}$.

- NCCCs are a weighted sum of processing and communication time. Similarly as for the simulated time metric, each agent a_i maintains an NCCCs counter c_i, which is initialized to 0. Every time a_i performs a constraint check (i.e., an evaluation for a partial assignment) it increments its current counter c_i by 1. Hence, it assigns $c_i = \max\{c_i, c_j + D\}$ when it receives a message from agent

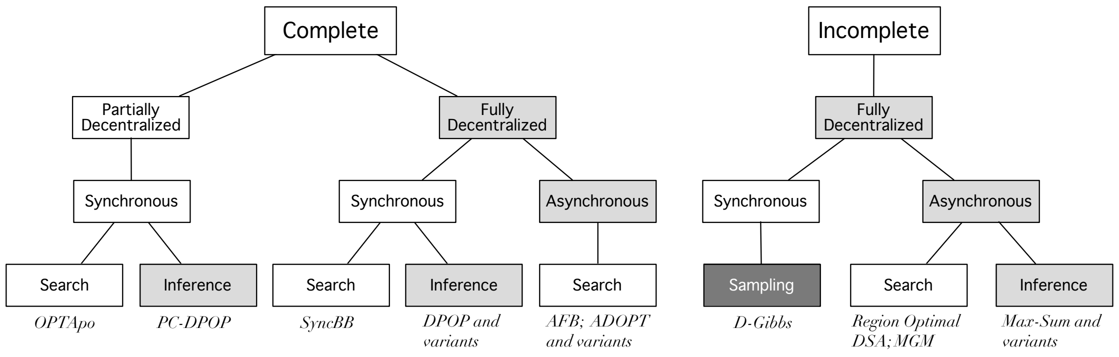


Figure 2.2: Classical DCOP Algorithm Taxonomy.

a_j to account for the time it takes to receive the message from a_j and for the transmission time of the message (D). The number of NCCCs of the algorithm is the largest counter value held by any agent.

In terms of network performance, DCOP algorithms are often evaluated by measuring the *network load* and the *message size* metrics, which are defined as follows:

- The *network load* refers to the total number of messages exchanged by the agents during the execution of the algorithm.
- The *message size* refers to the maximal size (typically is expressed in bytes) of the messages exchanged among the agents during the execution of the algorithm.

Complete Algorithms

SynchBB [56]. *Synchronous Branch-and-Bound* (SynchBB) is a complete, synchronous, search-based algorithm that can be considered as a distributed version of a branch-and-bound algorithm. It uses a complete ordering of the agents in order to extend a *Current Partial Assignment (CPA)* via a synchronous communication process. The CPA holds the assignments of all the variables controlled by all the visited agents, and, in addition, functions as a mechanism to propagate bound information. The algorithm prunes those parts of the search space whose solution quality is sub-optimal, by exploiting the bounds that are updated at each step of the algorithm. SynchBB agents space requirement and maximum size of message are in $O(n)$, while they require, in the worst case, to perform $O(d^m)$ number of operations. The network load is also in $O(d^m)$.

AFB [39]. *Asynchronous Forward Bounding* (AFB) is a complete, asynchronous, search-based algorithm that can be considered as the asynchronous version of SynchBB. In this algorithm, agents communicate their reward estimates, which in turn are used to compute bounds and prune the search space. In AFB, agents extend a CPA sequentially, provided that the lower bound on its reward does not exceed the global bound, that is, the reward of the best solution found so far. Each agent performing an assignment (the “assigning” agent) triggers asynchronous checks of bounds, by sending *forward* messages (referred to as *FB_CPA*) containing copies of the CPA to neighboring agents in the constraint graph that have not yet assigned their variables. The unassigned agents that receive a CPA, estimate the lower bound of the CPA, given their local view of the constraint graph. The cost estimates are returned back to the agent that originated the *forward* message, in *FB_ESTIMATE* messages. This assigning agent will receive these estimates asynchronously and aggregate them into an updated lower bound, which is used to prune the search space. If the updated lower bound exceeds the current upper bound, the agent initiates a backtracking phase. This process continues until the agent with lowest priority finds a complete solution, which is sent to all the agents. When the agent of highest priority exhausts all its value assignments, it broadcasts a termination message, assuming value from the best complete solution. As in SynchBB, the worst case complexity for network load and agent's operations is $O(d^m)$, while the size of messages and each agent's space requirement are in $O(n)$.

ADOPT [81]. *Asynchronous Distributed OPTimization (ADOPT)* is a complete, asynchronous, search-based algorithm that makes use of a DFS pseudo-tree ordering of the agents. The algorithm relies on maintaining, in each agent, lower and upper bounds on the solution reward for the subtree rooted at its node(s) in the DFS tree. Agents explore partial solutions in best-first order, that is, in increasing lower bound order. Agents use *COST* messages (propagated upwards in the DFS pseudotree) and *THRESHOLD* and *VALUE* messages (propagated downwards in the tree) to iteratively tighten the lower and upper bounds, until the lower bound of the minimum cost solution is equal to its upper bound. ADOPT agents store lower bounds as thresholds, which can be used to prune partial solutions that are provably sub-optimal. ADOPT agents need to maintain a *context* which stores the assignments of higher priority neighbors, and a lower bound and an upper bound for each domain value and child; thus, the space requirement for each agent is in $O(d(l+1))$, where $l = \max_{a_i \in \mathbf{A}} |N_{a_i}|$. Its worst case network load and agent complexity is $O(d^m)$, while its maximum message size is in $O(h)$. ADOPT has been extended in several ways. In particular, *BnB-ADOPT* [121, 50] uses a branch-and-bound method to reduce the amount of computation performed during search, and *ADOPT(k)* combines both ADOPT and BnB-ADOPT into an integrated algorithm [51]. There are also extensions that trade solution optimality for smaller runtimes [122], extensions that use more memory for smaller runtimes [123], and extensions that maintain soft arc-consistency [9, 8, 49, 47].

DPOP [96]. *Distributed Pseudo-tree Optimization Procedure (DPOP)* is a complete, synchronous, inference-based algorithm that makes use of a DFS pseudo-tree ordering of the agents. It involves three phases:

- *Pseudo-tree construction phase:* In the first phase, the agents order themselves into a DFS pseudo-tree.
- *Utility propagation phase:* In the second phase, each agent, starting from the leaves of the pseudo-tree, aggregates the rewards in its subtree for each value combination of variables in its separator.[2] The aggregated rewards are encoded in a *UTIL* message, which is propagated from children to their parents, up to the root.
- *Value propagation phase:* In the third phase, each agent, starting from the root of the pseudo-tree, selects the optimal values for its variables. The optimal values are calculated based on the UTIL messages received from the agent's children and the *VALUE* message received from its parent. The

[2]The separator of a_i contains all ancestors of a_i in the pseudo-tree (through tree edges or back edges) that are connected to a_i or one of its descendants.

VALUE messages contain the optimal values of the agents and are propagated from parents to their children, down to the leaves of the pseudo-tree.

Thus, DPOP generates a number of messages that is in $O(m)$. However, the size of the messages and the agent's space requirement are exponential in the induced width of the pseudo-tree: $O(d^{w^*})$. Finally, the number of operations performed by DCOP agents is in the order complexity of $O(d^{w^*+z})$, with $z = \max_{a_i \in \mathbf{A}} |L_i|$. DPOP has also been extended in several ways to enhance its performance and capabilities. O-DPOP and MB-DPOP trade runtimes for smaller memory requirements [97, 99], A-DPOP trades solution optimality for smaller runtimes [95], SS-DPOP trades runtime for increased privacy [42], PC-DPOP trades privacy for smaller runtimes [100], H-DPOP propagates hard constraints for smaller runtimes [63], BrC-DPOP enforces branch consistency for smaller runtimes [35], and ASP-DPOP is a declarative version of DPOP that uses Answer Set Programming [67].

OptAPO [74]. *Optimal Asynchronous Partial Overlay (OptAPO)* is a complete, synchronous, search-based algorithm. It trades agent privacy for smaller runtimes through partial centralization. It employs a cooperative mediation schema, where agents can act as mediators and propose value assignments to other agents. In particular, agents check if there is a conflict with some neighboring agent. If a conflict is found, the agent with the highest priority acts as a mediator. During mediation, OptAPO solves subproblems using a centralized branch-and-bound-based search, and when solutions of overlapping subproblems still have conflicting assignments, the solving agents increase the centralization to resolve them. By sharing their knowledge with centralized entities, agents can improve their local decisions, reducing the communication costs. For instance, the algorithm has been shown to be superior to ADOPT on simple combinatorial problems. However, it is possible that several mediators solve overlapping problems, duplicating efforts [100], which can be a bottleneck especially for dense problems. The worst case agent complexity is in $O(d^n)$, as an agent might solve the entire problem. The agent space requirement is in $O(nd)$, as a mediator agent needs to maintain the domains of all the variables involved in the mediation section, while the message size is in the order of $O(d)$. The network load decreases with the amount of partial centralization required, however, its worst case order complexity is exponential in the number of agents $O(d^m)$. The original version of OptAPO has been shown to be incomplete [45], but a complete variant has been proposed [45].

Incomplete Algorithms

Max-Sum [31]. *Max-Sum* is an incomplete, asynchronous, inference-based algorithm based on belief propagation. It operates on factor graphs by performing a marginalization process of the reward functions, and optimizing the rewards for each given variable. This process is performed by recursively propagating messages between variable nodes and functions nodes. The value assignments take into account their impact on the marginalized reward function. Max-Sum is guaranteed to converge to an optimal solution in acyclic graphs, but convergence is not guaranteed in cyclic graphs. Nevertheless, it has been shown to often converge in practice. Max-Sum has also been extended in several ways to improve it. Bounded Max-Sum is able to bound the quality of the solutions found by removing a subset of edges from a cyclic DCOP graph to make it acyclic, and by running Max-Sum to solve the acyclic problem [104], Improved Bounded Max-Sum improves on the error bounds [105], and Max-Sum_AD guarantees convergence in acyclic graphs through a two-phase value propagation phase [133]. Max-Sum and its extensions have been successfully employed to solve a number of large scale, complex MAS applications ([104, 79]).

Region Optimal [91]. *Region-optimal* algorithms are incomplete, synchronous, search-based algorithms that allow users to specify regions of the constraint graph (e.g., regions with a maximum size of k agents [91], t hops from each agent [58], or a combination of both size and hops [117]) and solve the subproblem within each region optimally. The concept of k-optimality is defined with respect to the number of agents whose assignments conflict, whose set is denoted by $\mathbf{c}(\sigma, \sigma')$, for two assignments σ and σ'. The deviating cost of σ with respect to σ', denoted by $\Delta(\sigma, \sigma')$, is defined as the difference of the aggregated reward associated to the assignment σ ($F(\sigma)$) minus the reward associated to σ' ($F(\sigma')$). An assignment σ is k-optimal if $\forall \sigma' \in \Sigma$, such that $|\mathbf{c}(\sigma, \sigma')| \leq k$, we have that $\Delta(\sigma, \sigma') \geq 0$. In contrast, the concept of t-distance emphasizes the number of hops from a central agent a of the region $\Omega_t(a)$, that is the set of agents which are separated from a by at most t hops. An assignment σ is t-distance optimal if, $\forall \sigma' \in \Sigma$, $F(\sigma) \geq F(\sigma')$ with $\mathbf{c}(\sigma, \sigma') \subseteq \Omega_t(a)$, for any $a \in \mathbf{A}$. The *Distributed Asynchronous Local Optimization* (DALO) simulator provides a mechanism to coordinate the decision of local groups of agents based on the concepts of k-optimality and t-distance [58]. The quality of the solutions found is bounded by a function of k or t [117].

MGM [72]. The *Maximum Gain Message (MGM)* is an incomplete, synchronous, search-based algorithm that performs a distributed local search. Each agent starts by assigning a random value to each of its variables. Then, it sends this information to all its neighbors. Upon receiving the values of its neighbors, it calculates the maximum gain in reward if it changes its value and sends this information to all its neighbors as well. Upon receiving the gains of its neighbors, it changes its value if its gain is the largest among its neighbors. This process repeats until a termination condition is met.

DSA [126]. The *Distributed Stochastic Algorithm (DSA)* is an incomplete, synchronous, search-based algorithm that is similar to MGM, except that each agent does not send its gains to its neighbors and it does not change its value to the value with the maximum gain. Instead, it decides stochastically if it takes on the value with the maximum gain or other values with smaller gains. This stochasticity allows DSA to escape from local minima. Similarly to MGM, it repeats until a termination condition is met.

D-Gibbs [84]. The *Distributed Gibbs* (D-Gibbs) algorithm is an incomplete, synchronous, sampling-based algorithm that extends the Gibbs sampling process [38] by tailoring it to solve DCOPs in a decentralized manner. The Gibbs sampling process is a centralized Markov chain Monte Carlo algorithm that can be used to approximate joint probability distributions. It generates a Markov chain of samples, each of which is correlated with previous samples. It does so by iteratively sampling one variable from the conditional probability distribution, assuming that all the other variables take their previously sampled values. This process continues for a fixed number of iterations, or until convergence, that is, the joint probability distribution approximated by the samples do not change. Once the joint probability distribution is found, one can identify a complete solution with the maximum likelihood. By mapping DCOPs to maximum a-posteriori estimation problems, probabilistic inference algorithms like Gibbs sampling can be used to solve DCOPs.

2.1.5 Relevant Uses

The classical DCOP model is capable of representing a wide range of MAS applications, especially those where agents in a team need to work cooperatively to achieve a single goal in a static, deterministic, and fully observable environment. Exploring the domain structural properties, as well as understanding the requirements of the problem designer, is crucial to design and apply effective DCOP algorithms. When an optimal solution is required, then a complete algorithm can be used to solve the problem. However, if particular assumptions can be made on the problem structure, more efficient solutions can be adopted. For instance, if the constraint graph of the DCOP is always a tree (i.e., it has no cycles) then an incomplete inference-based algorithm, like Max-Sum, is sufficient to guarantee the optimality of the solution found.

Complete algorithms are often unsuitable for tackling large-scale problems, due to their exponential requirements in time or memory. In contrast, incomplete algorithms are more appropriate to rapidly find solutions, at the cost of sacrificing optimality. The communication requirements also need to be taken into account. For example, when communication is unreliable, it is not recommended to employ search-based solutions, such as ADOPT or AFB, where communication requirements are exponential in the size of the problem. In contrast, inference-based algorithms are more reliable in the presence of uncertain communication networks as they, in general, require only a linear number of messages to complete their computations.

A popular application that is often referenced in the classical DCOP literature is the *Distributed Multi-Event Scheduling* (also known as *Meeting Scheduling*) [73]. It captures generic scheduling problems where one wishes to schedule a set of events within a time range. Each event is defined by (i) the time required to complete the event, (ii) the resources required to complete the event, and (iii) the cost of using such resources at a given time. A scheduling conflict occurs if two events with at least one common resource are scheduled in overlapping time slots. The goal is to maximize the overall reward, defined as the net gain between the opportunity benefit and opportunity cost of scheduling various events.

2.2 Overview of DCOP Extensions

The DCOP model has undergone a process of continuous evolution to capture diverse characteristics of agents behavior and the environment in which they operate. We propose a classification of DCOP models from a Multi-Agent Systems perspective, that accounts for the different assumptions made about the behavior of agents and their interactions with the environment. The classification is based on the following elements (summarized in Table 2.1):

ELEMENT		CHARACTERIZATION	
AGENT(S)	BEHAVIOR	Deterministic	Stochastic
	KNOWLEDGE	Total	Partial
	TEAMWORK	Cooperative	Competitive
ENVIRONMENT	BEHAVIOR	Deterministic	Stochastic
	EVOLUTION	Static	Dynamic

Table 2.1: DCOP classification elements.

- **Agent Behavior:** This parameter captures the stochastic nature of the effects of an action being executed. In particular, we distinguish between *deterministic* and *stochastic* effects.
- **Agent Knowledge:** This parameter captures the knowledge of an agent about its own state and the environment—distinguishing between *total* and *partial* knowledge.

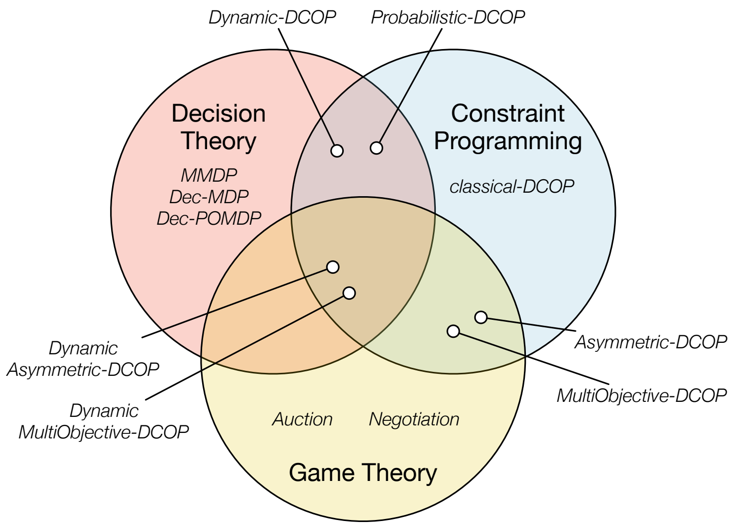


Figure 2.3: DCOPs within a MAS perspective.

- **Agent Teamwork:** This parameter characterizes the approach undertaken by (teams of) agents to solve a distributed problem. It can be either a *cooperative* resolution approach or a *competitive* resolution approach. In the former class, all agents cooperate to achieve a common goal (i.e., optimize a utility function). In the latter class, each agent (or team of agents) seeks to achieve its own individual goal.
- **Environment Behavior:** This parameter captures the exogenous properties of the environment. For example, it is possible to distinguish between *deterministic* and *stochastic* responses of the environment to the execution of an action.
- **Environment Evolution:** This parameter captures whether the DCOP is *static* (i.e., it does not change over time) or *dynamic* (i.e., it changes over time).

Figure 2.3 illustrates a categorization of the DCOP models proposed to date from a MAS perspective. In particular, we focus on the DCOP models proposed at the junction of *Constraint Programming (CP)*, *Game Theory (GT)*, and *Decision Theory (DT)*. The *classical* DCOP model is directly inherited from CP and characterized by a static model, a deterministic environment and agent behavior, total agent knowledge, and with cooperative agents. Concepts from auctions and negotiations, traditionally explored in GT, have influenced the DCOP framework, leading to Asymmetric DCOPs, which has asymmetric agent payoffs, and Multi-Objective DCOPs. The DCOP framework has borrowed fundamental DT concepts related to modeling uncertain and dynamic environments, resulting in models like Probabilistic DCOPs and Dynamic DCOPs. Researchers from the DCOP community have also designed solutions that inherit from all of the three communities.

In the next subsections, we will describe the different DCOP frameworks which extend the *classical* DCOP model. We focus on a categorization based on three dimensions: *Agent knowledge*, *environment behavior*, and *environment evolution*. We assume a *deterministic agent behavior*, *fully cooperative agent teamwork*, and *total agent knowledge* (unless otherwise specified), as they are, by far, common assumptions adopted by the DCOP community. The DCOP models associated to such categorization are summarized in Table 2.2. The bottom-right entry of the table is left empty, indicating a promising model with dynamic and uncertain environments that, to the best of our knowledge, has not been explored yet.

		Environment Behavior	
Environment Evolution		Deterministic	Stochastic
	Static	*classical-DCOP*	*probabilistic-DCOP*
	Dynamic	*dynamic-DCOP*	—

Table 2.2: DCOPs Models.

There has been only a modest amount of effort in modeling the different aspects of teamwork within the DCOP community.

2.2.1 Asymmetric DCOPs

Asymmetric DCOPs [44] are used to model multi-agent problems where two variables in the scope of the same reward function can receive different rewards from each other. Such a problem cannot be naturally represented by classical DCOPs, which require that all variables in the scope of the same reward function receive the same rewards as each other.

Definition An *Asymmetric DCOP* is defined by a tuple $\langle \mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha \rangle$, where $\mathbf{A}, \mathbf{X}, \mathbf{D}$ and α are defined as in section 2.1.2, and each $f_i \in \mathbf{F}$ is defined as: $f_i : \times_{x_j \in \mathbf{x}^i} D_j \times \alpha(f_i) \to (\mathbb{R}^+ \cup \{\perp\})$. In other words, an Asymmetric DCOP is a DCOP where the reward that an agent obtains from a reward function may differ from the reward another agent obtains from the same reward function.

As rewards for participating agents may differ from each other, the goal in Asymmetric DCOPs is also different than the goal in classical DCOPs. Given a reward function $f_j \in \mathbf{F}$ and complete solution σ, let $f_j(\sigma, a_i)$ denote the reward obtained by agent a_i from reward function f_j with solution σ. Then, the goal in Asymmetric DCOPs is to find the complete solution σ^*:

$$\sigma^* = \operatorname*{argmax}_{\sigma \in \Sigma} \sum_{f_j \in \mathbf{F}} \sum_{a_i \in \alpha(f_j)} f_j(\sigma_{\mathbf{x}^j}, a_i) \tag{2.2}$$

Relation to Classical DCOPs One way to solve MAS problems with asymmetric rewards via classical DCOPs is through the *Private Event As Variables* (PEAV) model [72]. It can capture asymmetric rewards by introducing, for each agent, as many "mirror" variables as the number of variables held by neighboring agents. The consistency with the neighbors' state variables is imposed by a set of equality constraints. However such formalism suffers from scalability problems, as it may result in a significant increase in the number of variables in a DCOP. In addition, Grinshpoun *et al.* showed that most of the existing incomplete classical DCOP algorithms cannot be used to effectively solve Asymmetric DCOPs, even when the problems are reformulated through the PEAV model [44]. They show that such algorithms are unable to distinguish between different solutions that satisfies all hard constraints, resulting in a convergence to one of those solutions and the inability to escape that local optimum. Therefore, it is important to generate ad-hoc algorithms to solve Asymmetric DCOPs.

2.2.2 Multi-Objective DCOPs

Multi-objective optimization (MOO) [78, 75] aims at solving problems involving more than one objective function to be optimized simultaneously. In a MOO problem, optimal decisions need to accommodate conflicting objectives. Examples of MOO problems include optimization of electrical power generation in a power grid while minimizing emission of pollutants and minimization of the costs of buying a vehicle while maximizing comfort. *Multi-objective DCOPs* extend MOO problems and DCOPs.

A *Multi-objective DCOP* (MO-DCOP) is defined by a tuple $\langle \mathbf{A}, \mathbf{X}, \mathbf{D}, \vec{\mathbf{F}}, \alpha \rangle$, where $\mathbf{A}, \mathbf{X}, \mathbf{D}$, and α are defined as in section 2.1.2, and $\vec{\mathbf{F}} = [F_1, \ldots, F_h]^T$ is a vector of multi-objective functions, where each F_i is a set of optimization functions f_j defined as in section 2.1.2. For a solution σ of a MO-DCOP, let the *reward* for σ according to the i^{th} multi-objective optimization function set F_i $(1 \leq i \leq h)$ be

$$F_i(\sigma) = \sum_{f_j \in F_i} f_j(\sigma_{\mathbf{x}^j}) \tag{2.3}$$

The goal of a MO-DCOP is to find an assignment σ^*, such that:

$$\sigma^* = \operatorname*{argmax}_{\sigma \in \Sigma} \vec{\mathbf{F}}(\sigma) = \operatorname*{argmax}_{\sigma \in \Sigma} [F_1(\sigma), \ldots, F_h(\sigma)]^T \tag{2.4}$$

where $\vec{\mathbf{F}}(\sigma)$ is a *reward vector* for the MO-DCOP. A solution to a MO-DCOP involves the optimization of a set of partially-ordered assignments. Note that we consider, in the above definition, point-wise comparison of vectors—i.e., $\vec{\mathbf{F}}(\sigma) \geq \vec{\mathbf{F}}(\sigma')$ if $F_i(\sigma) \geq F_i(\sigma')$ for all $1 \leq i \leq h$. Typically, there is no single global solution where all the objectives are optimized at the same time. Thus, solutions of a MO-DCOP are characterized by the concept of *Pareto optimality*, which can be defined through the concept of *dominance*:

Definition 1 (Dominance). *A complete solution $\sigma \in \Sigma$ is* dominated *by a complete solution $\sigma^* \in \Sigma$* iff *$\vec{\mathbf{F}}(\sigma^*) \geq \vec{\mathbf{F}}(\sigma)$ and $F_i(\sigma^*) > F_i(\sigma)$ for at least one F_i.*

Definition 2 (Pareto Optimality). *A complete solution $\sigma^* \in \Sigma$ is* Pareto optimal iff *it is not dominated by any other complete solution.*

Therefore, a solution is Pareto optimal iff there is no other solution that improves at least one objective function without deteriorating the reward of another function. Another important concept is the *Pareto front*:

Definition 3 (Pareto Front). *The* Pareto front *is the set of all reward vectors of all Pareto optimal solutions.*

Solving an MO-DCOP is equivalent to finding the Pareto front. Even for tree-structured MO-DCOPs, the size of the Pareto front may be exponential in the number of variables.[3] Thus, multi-objective algorithms often provide solutions that may not be Pareto optimal but may satisfy other criteria that are significant for practical applications.

2.2.3 Probabilistic DCOPs

So far, we have discussed DCOP models that caputre MAS problems in environments that are static and deterministic. However, many real-world applications are characterized by environments with stochastic

[3] In the worst case, every possible solution can be a Pareto optimal solution.

behavior. In other words, there are exogenous events that can influence the outcome of agent actions. For example, weather conditions or the state of a malfunctioning device can affect the reward of agent actions. To cope with such scenarios, researchers have introduced *Probabilistic DCOP (P-DCOP)* models, where the uncertainty in the state of the environment is modeled through stochasticity in the reward functions. With respect to our categorization, in the P-DCOP model the agents are completely cooperative and they have deterministic behavior. Additionally, the environment is static and stochastic. While a large body of research has focused on problems where agents have *total knowledge*, we will discuss a subclass of P-DCOPs where the agents' knowledge of the environment is limited, and agents must balance *exploration* of the unknown environment and the *exploitation* of the known rewards.

A common strategy to model uncertainty is to augment the outcome of the reward functions with a stochastic character [4, 110, 83]. Another method is to introduce additional random variables to the reward functions, which simulate exogenous uncontrollable traits [68, 69, 119]. To cope with such a variety, we introduce the *Probabilistic DCOP (P-DCOP)* model, which generalizes the proposed models of uncertainty. A P-DCOP is defined by a tuple $\langle \mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha, \mathcal{I}, \Omega, \mathbf{P}, \mathcal{U}, \mathcal{E} \rangle$, where $\mathbf{A}$ and $\mathbf{D}$ are defined as in section 2.1.2. In addition,

- $\mathbf{X}$ is a mixed set of decision variables and random variables.
- $\mathcal{I} = \{r_1, \dots, r_q\} \subseteq \mathbf{X}$ is a set of *random variables* modeling uncontrollable stochastic events, such as weather or a malfunctioning device.
- $\mathbf{F}$ is the set of reward functions, each defined over a mixed set of decision variables and random variables, and such that each value combination of the decision variables on the reward function, results in a probability distribution. As a result, the local value assignment $\sigma_{\mathbf{x}^i \setminus \mathcal{I}}$, given an outcome for the random variables involved in f_i, is itself a random variable.
- $\alpha : \mathbf{X} \setminus \mathcal{I} \to \mathbf{A}$ is a mapping from decision variables to agents. Notice that random variables are not controlled by any agent, as their outcomes do not depend on the agents' actions.
- $\Omega = \{\Omega_1, \dots, \Omega_q\}$ is the (possibly discrete) set of events for the random variables (e.g., different weather conditions or stress levels a device is subjected to) such that each random variable $r_i \in \mathcal{I}$ takes values in Ω_i. In other words, Ω_i is the domain of random variable r_i.
- $\mathbf{P} = \{p_1, \dots, p_q\}$ is a set of probability distributions for the random variables, such that $p_i : \Omega_i \to [0, 1] \subseteq \mathbb{R}$, assigns a probability value to an event for r_i, and $\int_{\omega \in \Omega_i} p_i(\omega)\, d\omega = 1$, for each random variable $r_i \in \mathcal{I}$.
- $\mathcal{U}$ is a *utility function* from random variables to random variables, that ranks different outcomes based on the decision maker preferences. This function is needed when the reward functions have uncertain outcomes, and thus these distribution are not readily comparable.
- $\mathcal{E}$ is an *evaluator function* from random variables to real values, that, given an assignment of values to the decision variables, summarizes the distribution of the aggregated reward functions.

The goal in a P-DCOP is to find a complete solution σ^*, that is, an assignment of values to all the decision variables, such that:

$$\sigma^* = \operatorname*{argmax}_{\sigma \in \Sigma} \ \mathcal{E}\left[\mathcal{U}\left(\sum_{f_i \in \mathbf{F}} f_i(\sigma_{\mathbf{x}^i \setminus \mathcal{I}}) \right) \right] \tag{2.5}$$

In other words, agents attempt to maximize the utility of the cumulative reward functions of the P-DCOP, with respect to the evaluator function $\mathcal{E}$.

The probability distribution over the domain of random variables $r_i \in \mathcal{I}$ is called a *belief*. An assignments of all random variables in $\mathcal{I}$ describes a (possible) *scenario* governed by the environment. As the random variables are not under the control of the agents, they *act* independently of the decision variables. Specifically, their beliefs are drawn from probability distributions. Furthermore, they are assumed to be independent of each other and, thus, they model independent sources of exogenous uncertainty.

The utility function $\mathcal{U}$ enables us to compare the uncertain reward outcomes of the reward functions. In general, the utility function is non-decreasing, that is, the higher the reward, the higher the utility. However, the utility function should be defined for the specific application of interest. For example, in farming, the utility increases with the amount of produce harvested. However, farmers may prefer a smaller but highly certain amount of produce harvested over a larger but highly uncertain and, thus, risky outcome. The evaluation function $\mathcal{E}$ is used to summarize in one criterion the rewards of a given assignment that depends on the random variables. A possible evaluation function is the *expectation* function: $\mathcal{E}[\cdot] = \mathbb{E}[\cdot]$.

2.2.4 Dynamic DCOPs

Within a real-world MAS application, agents often act in dynamic environments that evolve over time. For instance, in a disaster management search and rescue scenario, new information (e.g., the number of victims in particular locations, or priorities on the buildings to evacuate) typically becomes available in an incremental manner. Thus, the information flow modifies the environment over time. To cope with such requirement, researchers have introduced the *Dynamic DCOP (D-DCOP)* model, where reward functions can change during the problem solving process, agents may fail, and new agents may be added to the DCOP being solved. With respect to our categorization, in the D-DCOP model, the agents are completely cooperative and they have deterministic behavior and total knowledge. On the other hand, the environment is dynamic and deterministic.

Definition The *Dynamic DCOP (D-DCOP)* model is defined as a sequence of classical DCOPs: $\mathcal{D}_1, \ldots, \mathcal{D}_T$, where each $\mathcal{D}_t = \langle \mathbf{A}^t, \mathbf{X}^t, \mathbf{D}^t, \mathbf{F}^t, \alpha^t \rangle$ is a DCOP, representing the DCOP at time step t, for $1 \leq t \leq T$. The goal in a D-DCOP is to solve optimally the DCOP at each time step. We assume that the agents have total knowledge about their current environment (i.e., the current DCOP), but they are unaware of changes to the problem in future time steps.

In a dynamic system, agents are required to adapt as fast as possible to environmental changes. *Stability* [28, 115] is a core algorithmic concept, where an algorithm seeks to minimize the number of steps that it requires to converge to a solution each time the problem changes. In such a context, these converged solutions are also called *stable solutions*. *Self-stabilization* is a related concept derived from the the area of fault-tolerance:

Definition 4 (Self-stabilization)**.** *A system is* self-stabilizing *if and only if the following two properties hold:*

- Convergence: *The system reaches a stable solution in a finite number of steps, starting from any given state.*
- Closure: *The system remains in a stable solution, provided that no changes in the environment happens.*

An extension of the concept of self-stabilization is that of *super-stabilization* [29], which focuses on stabilization after topological changes. In the context of D-DCOPs, differently from self-stabilizing algorithms, where convergence after a single change in the constraint graph can be as slow as the convergence from an arbitrary starting state, super-stabilizing algorithms take special care of the time required to adapt to a single change in the constraint graph.

2.3 Overview of General Purpose Graphical Processing Units

Since in this dissertation we investigate the use of graphic cards to speed up the resolution approach of several classes of DCOP algorithm, we provide next an overview of such devices.

Modern *General Purpose Graphics Processing Units* (GPGPUs) are true multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy to support graphical processing (e.g., DirectX and OpenGL APIs). NVIDIA's *Compute Unified Device Architecture* (CUDA) [108] aims at enabling the use of the multiple cores of a graphic card to accelerate general (non-graphical) applications by providing programming models and APIs that enable the full programmability of the GPGPU. This movement allowed programmers to gain access to the parallel processing capabilities of a GPGPU without the restrictions of graphical APIs. In this dissertation, we will consider the CUDA programming model proposed by NVIDIA. The underlying conceptual model of parallelism supported by CUDA is *Single-Instruction Multiple-Threads* (SIMT), where the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread. The computational model supported by CUDA is *Single-Instruction Multiple-Data* (SIMD), where multiple threads perform the same operation on multiple data points simultaneously. CUDA's architectural model is represented in Figure 2.4.

2.3.1 Hardware Architecture

A GPGPU is a massive parallel architecture with thousands of computing cores. Different GPGPUs are distinguished by the number of cores, their organization, and the amount of memory available. A GPGPU is constituted by a series of *Streaming MultiProcessors* (SMs), whose number depends on the specific characteristics of each class of GPGPU. For example, the Fermi architecture provides 16 SMs, as illustrated in Figure 2.4 (left). Each SM contains from 8 to 32 computing cores, each of which incorporate an ALU and a floating-point processing unit. Each GPGPU provides access to both on-chip memory and off-chip memory, used in different contexts which we will introduce below.

2.3.2 Logical Architecture

Figure 2.4 (right) shows a typical CUDA logical architecture. A CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the *host*) and parts meant for parallel execution on the GPGPU (referred as the *device*). A parallel computation is described by a collection of *kernels*, where each kernel is a function to be executed by several threads. To facilitate the mapping of the threads to the data structures being processed: threads are organized in a 3-dimensional structure (called *block*), and blocks themselves are organized in 2-dimensional tables (called *grids*). When mapping a kernel to a specific GPGPU, CUDA schedules blocks (coarse-grain parallelism) on the SMs for execution.

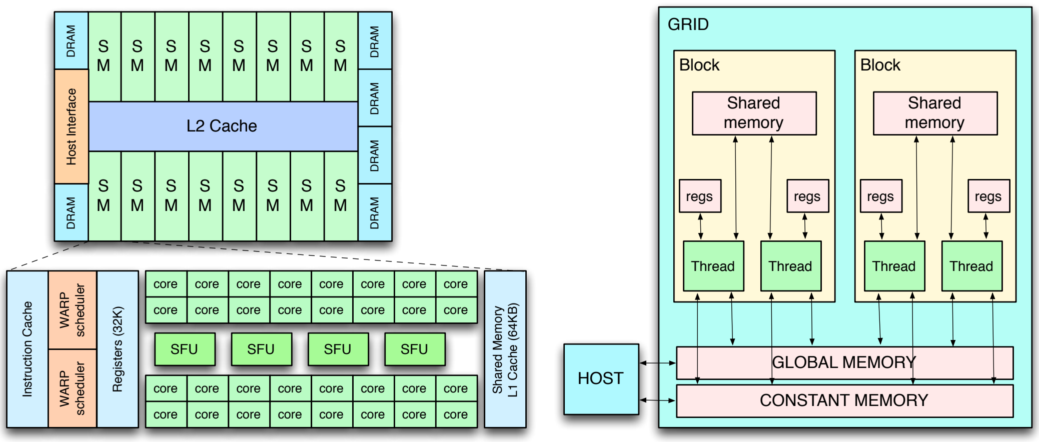


Figure 2.4: Fermi Hardware Architecture (left) and CUDA Logical Architecture (right)

Each SM schedules the threads in a block (fine-grain parallelism) on its computing cores in chunks of threads called *warps* (typically composed of 32 threads), which is the smallest work unit on the device. CUDA kernels may involve thousands of blocks and threads, thus this organization allows group of threads in a block to use the computing resources, while other threads of the same block might be waiting for information (e.g., completing a slow memory request).

The kernel, invoked by the host, is executed by the device and it is written in standard C/C++ code. The number of running blocks (gridDim), the number of threads of each block (blockDim), and the amount of shared memory in bytes (nbytes) is specified by the kernel call that is invoked on the host code with the following syntax:

$$\text{Kernel} <<< \text{gridDim, blockDim, nbytes} >>> (\text{arg}_1, \ldots, \text{arg}_n);$$

In order to perform a computation on the GPGPU, it is possible to move the data from the host memory to the device memory and vice versa. By using the specific identifier of each block (blockIdx, which provides the x, y coordinates of the block in the grid), its dimension (blockDim) and the identifier of each thread (threadIdx, which provides the x, y, z coordinates for the thread within the block), it is possible to differentiate both the data read by each thread and code to be executed. These variables are always accessible within kernel functions. The organization of the data in data structures and data access patterns play a fundamental role to the efficiency of the GPGPU computation. In particular, since the computational model is (SIMD), it is important that each thread in a warp executes the same branch of execution. When this condition is not satisfied (e.g., two threads execute different branches of a conditional construct) the degree of concurrency typically decreases, as the execution of threads performing separate control flows can be serialized. This is referred to as *branch divergence*, a phenomenon which has been intensely analyzed within the *High Performance Computing* (HPC) community [52, 18, 27].

2.3.3 Hierarchical Memory Organization

GPGPU and CPU devices are, in general, separate hardware units with physically distinct memories connected by a system bus. Thus, in order for the device to execute some computation invoked by the host and to return the results back to the caller, a data flow need to be enforced from the host memory to the device memory and vice versa.

The device memory architecture is quite different from that of the host, in that it is organized in several levels differing to each other for both physical and logical characteristics, such as location on the chip, access times, scope and lifetime of the data. In greater details, the device memory is organized into four different memory levels: **(1)** *registers*, **(2)** *shared memory*, **(3)** *local memory*, **(4)** *global memory*, **(5)** *constant memory*, and **(6)** *texture memory*. The only two types of memory that actually reside on the GPGPU chip are registers and shared memory. Local, global, constant, and texture memory all reside off chip, with the difference that constant and texture memory are cached, while local and global memories are not.

The data stored in the registers and in the local memory has a thread lifetime and visibility, while shared memory data is visible to all threads within a block, and has thus the same lifetime of a block. This is invaluable because this type of memory enables threads to communicate and share data between one another. The data stored in the global memory has global visibility and lifetime, thus it is visible to all threads within the application (including the host), and lasts for the duration of the host allocation. Local memory is not a physical type of memory, but an abstraction of global memory. Its scope is local to the thread, but residing off-chip makes it expensive to access to it. Such memory is used to hold *automatic variables*. The compiler makes use of local memory when it determines that there is not enough register space to hold the variable. Constant memory is a read-only memory and can be used rather than global memory to reduce the required memory bandwidth, however, this performance gain can only be realized when a warp of threads read the same location.Similar to constant memory, texture memory is another variety of read-only memory on the device. When all reads in a warp are physically adjacent, using texture memory can reduce memory traffic and increase performance compared to global memory.

Apart from lifetime and visibility, different memories have also different dimension, bandwidth, and access times. A typical register access consumes zero clock cycles per instruction. However, delays can occur due to read after write dependencies and bank conflicts (up to 24 clock cycles of latency). The total amount of shared memory is 48KB, and 16KB are used for L1 cache. This size can be set to 16KB, 32KB or 48KB, with the remaining amount automatically used for L1 cache. Since shared memory can be accessed by all threads, potential bottlenecks may arise when many threads attempt to access it at the same time. To alleviate such issue, the shared memory is divided into 32 logical banks, with successive sections of memory mapped to successive banks. There are 32 threads in a warp and exactly 32 shared memory banks. Since each bank serves exactly one request per cycle, multiple simultaneous accesses to the same bank will result in *bank conflicts*. When there are no bank conflicts, shared memory performance is comparable to register memory. The constant memory is limited to 64KB and, texture and global memories are the slowest and largest memories accessible by the device, with access times ranging from 300 to 600 clock cycles. Constant and texture memory are beneficial for only very specific types of applications, where for instance data is organized in 2- or 3- dimensional arrays. Even if not cached, global accesses covering a contiguous 64 bytes data are fetched at once.

While it is relatively simple to develop correct CUDA programs (e.g., by incrementally modifying an existing sequential program), it is nevertheless challenging to design an efficient solution. Several factors are critical in gaining performance. The SIMT model requires active threads in a warp to be executing the same instruction – thus, diverging flow paths among threads may reduce the amount of actual concurrency. Memory levels have significantly different sizes and access times, different cache behaviors are applied to different memory levels, and various optimization techniques are used (e.g., accesses to consecutive global memory locations by contiguous threads can be *coalesced* into a single memory transaction). Thus, optimization of CUDA programs require a thorough understanding of the hardware characteristics of the GPGPU being used.

3 Exploiting the Structure of DCOPs from Problem Modeling

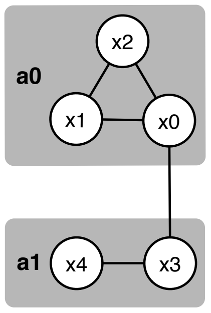
This chapter introduces a novel *Multi-Variable Agent (MVA)* DCOP decomposition technique which exploits co-locality of each agent's variables, allowing us to adopt efficient centralized techniques within each DCOP agent. Additionally, it reduces the amount of communication required in several classes of DCOP algorithms, and as we will show in section 5.3 it enables the use of hierarchical parallel models, such as those based on *GPGPUs*. Our experimental results, on both random graph and structured networks, show that this MVA decomposition outperforms non-decomposed DCOP algorithms, in terms of network load and scalability. Therefore, these results validate the hypothesis that one could exploit latent structure of DCOPs, embedded into their model, to speed up their resolution.

This chapter is organized as follows: We first discuss the motivation for our work in section 3.1. In section 3.2, we introduce our MVA decomposition, providing a description of how several classes of DCOP algorithms are automatically handled by such technique. We thus, discuss the theoretical properties, related to correctness, completeness, and agent and space complexity, associated with the use of this DCOP decomposition, in section 3.3, and present the experimental results in section 3.5. Finally we provide a discussion on the ability of our MVA decomposition technique to enable the use of hierarchical parallel models as byproduct, and conclude the chapter in section 3.6.

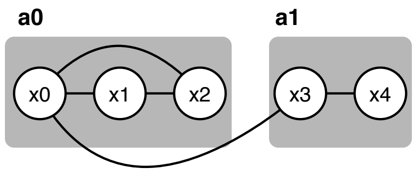
3.1 Motivations

The common resolution approach to DCOP solving is based on the assumption that each agent controls exclusively a single variable of the problem. However, modeling many real-world problems as DCOPs often require each agent to control a large number of variables. For instance, in a typical meeting scheduling problem, agents representing different organizations should handle multiple meetings. Figure 3.1(a) illustrates a scenario where agents control multiple variables, showing the constraint graph of a simple DCOP with 2 agents a_0 and a_1, where each variable can be assigned the values 0 or 1. Figure 3.1(c) shows the objective functions of the problem.

To cope with such restrictions, *reformulation techniques* are commonly adopted to transform a general DCOP into one where each agent controls exclusively one variable. There are two commonly used refor-



(a) Constraint Graph



(b) Linear Ordering

for $i < j$

x_i	x_j	Costs
0	0	7
0	1	10
1	0	2
1	1	3

(c) Cost Functions

Figure 3.1: Example DCOP.

mulation techniques [14, 125]: *(i) Compilation*, where each agent creates a new *pseudo-variable*, whose domain is the Cartesian product of the domains of all variables of the agent; and *(ii) Decomposition*, where each agent creates a *pseudo-agent* for each of its variables. While both techniques are relatively simple, they can be inefficient, as they ignore the structure present in the problem model. In compilation, the memory requirements for each agent grow exponentially with the number of variables that it controls. In decomposition, the DCOP algorithms will treat two pseudo-agents as independent entities, resulting in unnecessary computation and communication costs.

Figure 3.2 shows a snippet of the messages sent by the AFB (introduced in section 2.1.4) agents in our example DCOP of Figure 3.1 after a decomposition reformulation, where agent a_i^j is the pseudo-agent that controls variable x_j of agent a_i. We assume that the pseudo-agents are ordered as in Figure 3.1(b). AFB requires 98 messages between pseudo-agents controlled by different agents (i.e., actual agent-to-agent messages) and 60 messages between pseudo-agents controlled by the same agent (i.e., internal agent messages). Using such reformulation, each pseudo-agent has to obey the assumptions made in section 2.1.3, which apply to the agents' coordination process, even when the pseudo-agents associate to variables controlled by the same agent. This process thus, under-exploits the information regarding the global state of the variables controlled by an agent, both within the internal communication, and in the agent-to-agent knowledge propagation as the latter relates on the agents internal state. For instance, when using such decomposition technique, bound propagation in search-based algorithms cannot exploit co-locality of the variables within an agent; this results in decoupling the variable's domain information which may in turn result in propagating weak bounds.

Therefore, we hypothesize that by exploiting the information encoded in the distributed constraint optimization model DCOP algorithms can reduce the time of the resolution process, as well ensure a more efficient network load.

To validate this hypothesis we propose a DCOP *Multiple-Variable Agents* (MVA) problem decomposition that defines a clear separation between the distributed DCOP resolution and the centralized agent sub-problem resolution. This separation exploits co-locality of agent's variables, allowing the adoption of efficient centralized techniques to solve agent sub-problems. The distributed agents coordination problem is solved independently by the resolution of the agent sub-problems, and achieved by employing a global DCOP algorithm. Importantly, the proposed decomposition does not lead to any additional privacy loss. Furthermore, we show that the MVA framework naturally enables the use of different centralized and

Sender	Message Type	Receiver	Message Content
a_0^0	[CPA_MSG]	a_0^1	`[0 - - - -] (0)`
a_0^0	[FB_CPA]	$a_0^1, a_0^2, a_1^3, a_1^4$	`[0 - - - -] (0)`
a_0^1	[FB_ESTIMATE]	a_0^0	`(7)`
a_0^1	[CPA_MSG]	a_0^2	`[0 0 - - -] (7)`
a_0^0	[FB_CPA]	a_0^2, a_1^3, a_1^4	`[0 0 - - -] (7)`
a_0^2	[FB_ESTIMATE]	a_0^0	`(9)`
a_0^2	[FB_ESTIMATE]	a_0^1	`(14)`
a_0^2	[CPA_MSG]	a_1^3	`[0 0 0 - -] (21)`
a_0^2	[FB_CPA]	a_1^3, a_1^4	`[0 0 0 - -] (21)`
a_1^3	[FB_ESTIMATE]	a_0^0, a_0^1, a_0^2	`(7)`
a_1^3	[CPA_MSG]	a_1^4	`[0 0 0 0 -] (28)`
a_1^3	[FB_CPA]	a_1^4	`[0 0 0 0 -] (28)`
a_1^3	[FB_ESTIMATE]	a_0^0, a_0^1, a_0^2	`(2)`
a_1^4	[NEW_SOLUTION]	$a_0^0, a_0^1, a_0^2, a_1^3$	`[0 0 0 0 0] (35)`
...		...	...

Figure 3.2: Partial Trace of AFB after Decomposition.

distributed solvers in a hierarchical and parallel way [36, 37].

To illustrate the generality of the proposed framework we explore the use of two centralized solvers, *Depth-First Branch and Bound (DFBnB)* [11, 89] and *Gibbs Sampling* [38], to solve the agents' local subproblems. For the global coordination, we consider three representative DCOP algorithms: *Asynchronous Forward Bounding (AFB)* [39], as an example of a search algorithm, *Distributed Pseudo-tree Optimization Procedure (DPOP)* [96], as an example of an inference algorithm, and *Distributed Gibbs (D-Gibbs)* [84], as an example of a sampling algorithm.

3.2 MVA Decomposition

We now introduce our Multiple Variable Agent (MVA) decomposition for DCOPs. We first introduce some concepts employed by our decomposition, and hence describe the MVA framework.

3.2.1 Notation and Definitions

Given a DCOP $P = (\mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha)$, defined as in section 2.1.2, we introduce the following concepts.

Definition 5 (Local Variables)**.** *For each agent* $a_i \in \mathbf{A}$*,* $L_i = \{x_j \in \mathbf{X} \mid \alpha(x_j) = a_i\}$ *is the set of variables under the control of agent* a_i*, referred to as its* local *variables.*

Definition 6 (Boundary Variables)**.** *For each agent* $a_i \in \mathbf{A}$*,* $B_i = \{x_j \in L_i \mid \exists x_k \in \mathbf{X} \wedge \exists f_s \in \mathbf{F} : \alpha(x_k) \neq a_i \wedge \{x_j, x_k\} \subseteq \mathbf{x}^s\}$ *is the set of its* boundary *variables.*

In other words, a variable of an agent a_i is said *boundary* if it appears in the scope of an objective function which involves variables controlled by different agents. Note that, the actions to determine the

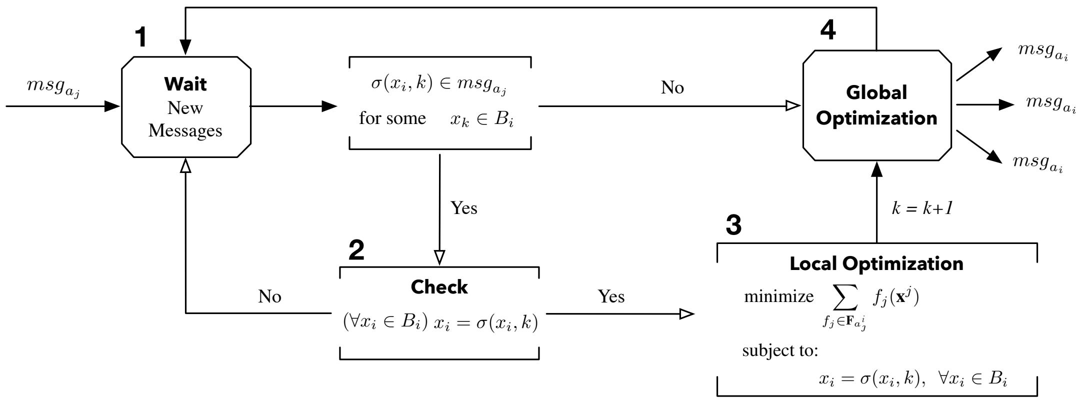

1
Wait
New Messages
msg_{a_j}
$\sigma(x_i, k) \in msg_{a_j}$
for some $x_k \in B_i$
No
Yes
2
Check
$(\forall x_i \in B_i)\ x_i = \sigma(x_i, k)$
No
Yes
3
Local Optimization
minimize $\sum_{f_j \in \mathbf{F}_{a_j}^i} f_j(\mathbf{x}^j)$
subject to:
$x_i = \sigma(x_i, k),\ \forall x_i \in B_i$
$k = k+1$
4
Global Optimization
msg_{a_i}
msg_{a_i}
msg_{a_i}

Figure 3.3: MVA Execution Flow Chart.

values of such variables are not dictated solely by the agent's decisions, but they require some coordination among different agents.

Definition 7 (Local Constraint Graph)**.** *For each agent* $a_i \in \mathbf{A}$*, its* local constraint graph $G_i = (L_i, E_{\mathbf{F}_{a_i}})$ *is a subgraph of the constraint graph* G_P*, where* $\mathbf{F}_{a_i} = \{f_j \in \mathbf{F} \mid \mathbf{x}^j \subseteq L_i\}$.

In other words, the local constraint graph of an agent a_i is the subgraph of G_P that includes the local variables of agent a_i and the constraints whose scopes include exclusively such local variables. In Figure 3.1(a), $L_0 = \{x_0, x_1, x_2\}$, $L_1 = \{x_3, x_4\}$, $B_0 = \{x_0\}$, $B_1 = \{x_3\}$.

We use $\sigma(x_i, k) \in D_i$ to denote the k^{th} value assignment to variable x_i.

Next, we describe how the MVA decomposition is applied to solve a general DCOP, exploiting the combination of decentralized DCOP algorithms, off-the-shelf centralized solvers, and their GPGPU parallel versions.

3.2.2 Description of the MVA Decomposition

In the MVA decomposition, a DCOP problem P is decomposed into $|\mathbf{A}|$ subproblems $P_i = (L_i, B_i, \mathbf{F}_{a_i})$, where P_i is associated to agent $a_i \in \mathbf{A}$. In addition to the decomposed problem P_i, each agent receives:

- The *global* DCOP algorithm P_G, which is common to all agents in the problem and defines the agent's coordination protocol and the behavior associated to the receipt of a message;
- The *local* algorithm P_L, which can differ between agents and is used to solve the agent's subproblem.

Figure 3.3 shows a flow chart illustrating the four conceptual phases in the execution of the MVA framework for each agent a_i:

- **Phase 1—Wait:** The agent waits for a message to arrive. If the received message results in a new value assignment $\sigma(x_r, k)$ for a boundary variable x_r of B_i, then the agent will proceed to Phase 2. If not, it will proceed to Phase 4.

- **Phase 2—Check:** The agent checks if it has performed a complete new assignment for all its boundary variables, indexed with $k \in \mathbb{N}$, which establishes an enumeration of the boundary variables' assignments. If it has, then the agent will proceed to Phase 3, otherwise it will return to Phase 1.

- **Phase 3—Local Optimization:** When a complete assignment is given, the agent passes the control to a local solver, which solves the following problem:

$$\text{Minimize}: \quad \sum_{f_j \in \mathbf{F}_{a_i}} f_j(\mathbf{x}^j) \tag{3.1}$$

$$\text{Subject to}: \quad x_r = \sigma(x_r, k) \quad \forall x_r \in B_i \tag{3.2}$$

  Solving this problem results in finding the best assignment for the agent's local variables given the particular assignment for its boundary variables. Notice that the local solver P_L is independent from the DCOP structure and it can be customized based on the agent's local requirements. Thus, agents can exploit a number of techniques for their local problem resolution. There exists a large number of off-the-shelf solvers developed through decades of research in various fields that can solve Constraint Optimization Problems. For example, such problems can often be formulated as linear programs and solved using solvers that have been honed by the operations research community [22, 23]; one can use constraint programming techniques, such as consistency maintenance procedures [26]; or they can be reformulated as optimization problems on graphical models [118] and solved using machine learning techniques [64, 40]. One can even exploit novel hardware platforms, such as GPGPUs, to parallelize such solvers [108, 17, 16].

  Once the agent solves its subproblem, it proceeds to Phase 4.

- **Phase 4—Global Optimization:** The agent processes the new assignment as established by the DCOP algorithm P_G, executes the necessary communications, and returns to Phase 1. The agents can execute these phases independently of one another because they exploit the co-locality of their local variables without any additional privacy loss, which is a fundamental aspect in DCOPs [43].

In addition, the local optimization process can operate on $m \geq 1$ combinations of value assignments of the boundary variables, before passing control to the next phase. This is the case when the agent explores m different assignments for its boundary variables in Phases 2 and 3. These operations are performed by storing the best local solution and their corresponding costs in a cost table of size m, which we call MVA_TABLE. Thus it can be seen as a cache memory. The minimum value of m depends on the choice of the global DCOP algorithm P_G. For example, for common search-based algorithms such as AFB, it is 1, while for common inference-based algorithms such as DPOP, it is exponential in size of the separator set.

Figures 3.4(b) and 3.4(c) show the MVA_TABLES of the two agents in our example DCOP with $m = 2$, and Figure 3.4(a) reports the constraint table for their boundary variables. Using the MVA decomposition, each agent computes only the necessary rows of the table on demand. Figure 3.5 shows the messages sent by agents in our example DCOP with the MVA framework. In total, AFB requires only 13 messages (compared to 98 messages with the decomposition reformulation) between agents. Additionally, since the local subproblem of each agent is solved using a local search engine, the agents do not need to send any internal agent messages (compared to 60 messages with the decomposition reformulation).

x_0	x_3	Costs
0	0	7
0	1	10
1	0	2
1	1	3

(a) Constraint Table of Boundary Variables

x_0	Best Local Solutions	Costs
0	$[x_1 = 1, x_2 = 0]$	19
1	$[x_1 = 1, x_2 = 1]$	7

(b) a_0's MVA_TABLE

x_3	Best Local Solutions	Costs
0	$[x_4 = 0]$	7
1	$[x_4 = 0]$	2

(c) a_1's MVA_TABLE

Figure 3.4: MVA_TABLES.

3.2.3 Local Optimization

We use *Depth First Search Branch and Bound* (DFBnB) and Gibbs as representative complete and incomplete algorithms for the local optimization process within each agent. DFBnB is correct and complete, thus, it does not affect correctness and completeness of the global complete DCOP algorithms used during agents coordination (see Theorem 1). In addition, it allows us to exploit the problem structure by bound propagation. Gibbs provides quality guarantees and can be used in combination with D-Gibbs to provide good approximated solutions (see Theorem 4).

Without loss of generality, in the following description, we assume that all variables $x_i \in L_i$ have the same domains, denoted to as D_i.

Depth First Search Branch and Bound

Depth First Search Branch and Bound (DFBnB) [11, 89] is a classic complete search algorithm that explores the variables' values in a depth-first order. DFBnB uses an upper bound α on the optimal final cost, whose initial value can be infinity. Starting at the root node (which corresponds to the first variable in the problem, according to a given ordering), DFBnB selects a value for the next variable in the order to examine next. When all the variables are assigned, meaning that DFBnB is exploring a leaf node, the algorithm revises the upper bound if the cost of the current solution is less than the current upper bound α. When DFBnB is exploring an internal node n, it compares the cost of the current partial solution with the current upper bound α. If such cost is greater than or equal to α, then any possible solution with the same partial assignments from the root node up to node n can be pruned. The reason is because node costs are non-decreasing along a path from the root, so that no descendent of a node n will have a cost smaller than n's cost. Otherwise, n is expanded, generating all its child nodes. The process continues until no more nodes can be expanded.

Sender	Message Type	Receiver	Message Content
a_0	[CPA_MSG]	a_1	[0 1 0 - -] (19)
a_0	[FB_CPA]	a_1	[0 1 0 - -] (19)
a_1	[FB_ESTIMATE]	a_0	(9)
a_1	[NEW_SOLUTION]	a_0	[0 1 0 0 0] (33)
a_1	[NEW_SOLUTION]	a_0	[0 1 0 1 0] (31)
a_1	[CPA_MSG]	a_0	[0 1 0 - -]
a_0	[CPA_MSG]	a_1	[1 1 0 - -] (7)
a_0	[FB_CPA]	a_1	[1 1 0 - -] (7)
a_1	[FB_ESTIMATE]	a_0	(4)
a_1	[NEW_SOLUTION]	a_0	[1 1 0 0 0] (16)
a_1	[NEW_SOLUTION]	a_0	[1 1 0 1 0] (12)
a_1	[CPA_MSG]	a_0	[1 1 0 - -]
a_0	[TERMINATE]	a_1	

Figure 3.5: Complete trace of MVA-AFB.

Gibbs Sampling

The Gibbs sampling algorithm [38] is a Markov chain Monte Carlo algorithm that can be used to approximate joint probability distributions. It generates a Markov chain of samples, each of which is correlated with previous samples. Suppose we have a joint probability distribution $P(z_1, z_2, \ldots, z_n)$ over n variables, which we would like to approximate. Algorithm 1 shows the pseudocode of the Gibbs algorithm, where each variable z_i^t represents the t-th sample of variable z_i. The algorithm first initializes z_i^0 to any arbitrary value of variable z_i (lines 1-3). Then, it iteratively samples z_i^t from the conditional probability distribution assuming that all the other $n-1$ variables take on their previously sampled values, respectively (lines 4-8). This process continues for a fixed number of iterations or until convergence, that is, the joint probability distribution approximated by the samples do not change. It is also common practice to ignore a number of samples at the beginning as it may not accurately represent the desired distribution. Once the joint probability distribution is found, one can easily identify that a complete solution with the maximum likelihood.

Algorithm 1: GIBBS($z_1, \ldots, z_n$)

```
1 for $i = 1$ to $n$ do
2     $z_i^0 \leftarrow$ INITIALIZE($z_i$)
3 for $t = 1$ to $T$ do
4     for $i = 1$ to $n$ do
5         $z_i^t \leftarrow$ SAMPLE($P(z_i \mid z_1^t, \ldots, z_{i-1}^t, z_{i+1}^{t-1}, \ldots, z_n^{t-1})$)
```

While the Gibbs algorithm is designed to solve the (maximum a posteriori) MAP estimation problem, it can also be used to solve DCOPs in a *centralized manner* by mapping MAP estimation problems to DCOPs [84]. If the probabilities in the MAP estimation problem is defined according to the DCOP utility functions as shown below

$$P(x_1, \dots, x_n) = \frac{1}{Z} \prod_{f_i \in \mathbf{F}} \exp[f_i(x_k \mid x_k \in S_i)] \tag{3.3}$$

$$= \frac{1}{Z} \exp\left[\sum_{f_i \in \mathbf{F}} f_i(x_k \mid x_k \in S_i)\right] \tag{3.4}$$

then a solution to the MAP estimation problem is also a solution to the DCOP.

3.3 Theoretical Results

In this section, we prove the correctness and completeness of the MVA framework when both the global DCOP algorithm P_G and the local algorithm P_L are correct and complete. We provide bounds for the additional space requirement and for the network load of the MVA framework with respect to the global DCOP algorithm adopted. Finally, we prove the equivalence of the MVA decomposed D-Gibbs sampling process and the (non MVA-decomposed) D-Gibbs algorithm.

Theorem 1. *The MVA framework with P_G and P_L is correct and complete if and only if P_G and P_L are both correct and complete.*

Proof. Let Ω^* be the set of complete optimal solutions of a DCOP instance P, and let us denote with $\Omega^*_{|S}$ as the set of all assignments for the variables in S that can be extended to a complete optimal solution for P. Given a solution $\mathbf{x}$ for the problem P, let us also denote to $\mathbf{x}_{|S}$ as for the projection of values of $\mathbf{x}$ to the variables of the set S.

Soundness: Let us prove the forward direction for soundness. Assume that the combination P_G and P_L with the MVA framework is correct and that it finds an optimal complete solution $\mathbf{x}^* \in \Omega^*$. Now assume that P_G is not correct. Then, an agent a_i might not explore the combination of values $\langle v^i_1, \dots, v^i_{b_i} \rangle \in \mathbf{x}^*$ for its boundary variables $x^i_j \in B_i$ $(j = 1, \dots, b_i)$, which contradicts the assumption that the MVA framework finds the optimal complete solution $\mathcal{V}^*$. Therefore, P_G is correct. The argument for P_L is similar to that of P_G. Assume that P_L is not correct. Therefore, an agent a_i might not explore the combination of values $\langle v^i_{b_i+1}, \dots, v^i_{l_i} \rangle$ for its non-boundary local variables $x^i_j \in L_i \setminus B_i$, $(j = b_i + 1, \dots, l_i)$, which contradicts the assumption that the MVA framework finds the optimal complete solution $\mathbf{x}^*$. Therefore, P_L is correct.

We now prove the backward direction for soundness. Assume that P_G and P_L are correct. Now assume that their combination within the MVA framework results in finding a solution $\mathbf{x} \notin \Omega^*$. If $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i} \in \Omega^*_{|\cup_{a_i \in \mathbf{A}} B_i}$, then for some agent a_i the combination of values $\mathbf{x}_{|L_i \setminus B_i}$ for its non-boundary local variables is such that $\mathbf{x}_{|L_i \setminus B_i} \notin \Omega^*_{|L_i \setminus B_i}$. This contradicts the assumption on the correctness of P_L. If $\mathbf{x} \in \Omega^*_{|\cup_{a_i \in \mathbf{A}} L_i \setminus B_i}$, then for some agent the combination of values $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i}$ for the problem boundary variables is such that $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i} \notin \Omega^*_{|\cup_{a_i \in \mathbf{A}} B_i}$. This contradicts the assumption on the correctness of P_G.

Completness: Let us prove the forward direction for completness. Assume that the combination P_G and P_L with the MVA framework is complete and that it finds all optimal complete solutions $\mathbf{x}^* \in \Omega^*$. Now assume that P_G is not complete. Then, an agent a_i might not explore the combination of values $\langle v^i_1, \dots, v^i_{b_i} \rangle \in \mathbf{x}^*$ for its boundary variables $x^i_j \in B_i$ $(j = 1, \dots, b_i)$, which contradicts the assumption that the MVA framework is complete. Therefore, P_G is complete. The argument for P_L is similar to that of P_G. Assume that P_L is not complete. Therefore, an agent a_i might not explore the combination of values $\langle v^i_{b_i+1}, \dots, v^i_{l_i} \rangle$ for its non-boundary local variables $x^i_j \in L_i \setminus B_i$, $(j = b_i + 1, \dots, l_i)$, which contradicts the assumption that the MVA framework is complete. Therefore, P_L is complete.

We now prove the backward direction for completness. Assume that P_G and P_L are complete. Now assume that there is a solution $\mathbf{x} \in \Omega^*$ that is not explored by the MVA framework. If $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i} \in \Omega^*_{|\cup_{a_i \in \mathbf{A}} B_i}$, then for some agent a_i the combination of values $\mathbf{x}_{|L_i \setminus B_i}$ for its non-boundary local variables is such that $\mathbf{x}_{|L_i \setminus B_i} \not\in \Omega^*_{|L_i \setminus B_i}$. This contradicts the assumption on the completeness of P_L. If $\mathbf{x} \in \Omega^*_{|\cup_{a_i \in \mathbf{A}} L_i \setminus B_i}$, then for some agent the combination of values $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i}$ for the problem boundary variables is such that $\mathbf{x}_{|\cup_{a_i \in \mathbf{A}} B_i} \not\in \Omega^*_{|\cup_{a_i \in \mathbf{A}} B_i}$. This contradicts the assumption on the completeness of P_G. □

Theorem 2. *The additional space requirement for the MVA framework is $O(M \cdot l)$, where M is the maximal number of rows of* MVA_TABLE *needed on demand by each agent a_i and $l = \max_{i \in \{j \mid a_j \in \mathbf{A}\}} |L_i|$.*

Proof. At each P_L invocation, for each row of MVA_TABLE needed on demand, each agent a_i maintains its value assignments for the boundary variables in $O(|B_i|)$ space and stores the local search results in $O(|L_i \setminus B_i|)$ space. Therefore, the total space needed is

$$\begin{aligned} m_i \cdot (O(|B_i| + |L_i \setminus B_i|) &= m_i \cdot O(|L_i|) \\ &= O(M \cdot l) \end{aligned}$$

□

Theorem 3. *The message requirement for the MVA framework is of the same order-complexity of that of P_G.*

Proof. As MVA emulates the message exchanging protocol adopted by P_G, its message requirements follows the same order-complexity, with respect to the number of agents of the DCOP. □

Note that this message complexity is for the worst case where all local variables are boundary variables. In problems with non-boundary local variables, the network load of search algorithms is often significantly smaller with the MVA decomposition (= $O(d^{|\mathbf{A}|})$) than with the Decomposition technique (= $O(d^{|\mathbf{X}|})$). See our example in Figures 3.2 and 3.5.

Theorem 4. *The sampling processes of both MVA-DG (= the MVA framework with D-Gibbs as P_G and Gibbs as P_L) and D-Gibbs converge to the same solution.*

Proof. To show that both algorithms converge to the same solution, we need to show that the transition matrices T^M and T^D, which describes the transition rules from state to state for MVA-DG and D-Gibbs, respectively, are equivalent. Assume that $T^M \neq T^D$ and consider the transition from a given state $\mathbf{z}^t$ to a state $\mathbf{z}^{t+1}$. In both algorithms, this transition depends on the sampling process for a variable $x_i \in \mathbf{X}$. We have the following two cases:

Case 1: $x_i \in B_i$ for some agent $a_i \in \mathbf{A}$. Since MVA-DG uses D-Gibbs as P_G, then both MVA-DG and D-Gibbs perform the same process to sample values for variable x_i. Thus, it trivially follows that $T^M = T^D$.

Case 2: $x_i \in L_i \setminus B_i$ for some agent $a_i \in \mathbf{A}$. By assumption, the new state $\hat{\mathbf{z}}^{t+1}$, produced by applying transition matrix T^M to $\mathbf{z}^t$, is such that $\hat{\mathbf{z}}^{t+1} \neq \mathbf{z}^{t+1}$. Note that during the Gibbs sampling process described by P_L, the value for the variable x_i is sampled according to the following:

$$P^{\text{MVA-DG}}[d] = P(x_i = d \mid x_l \in L_i \setminus \{x_i\})$$

while in D-Gibbs, it is sampled according to the following:

$$P^{\text{Gibbs}}[d] = P(x_i = d \mid x_l \in \mathbf{X} \setminus \{x_i\})$$

Since x_i shares constraints exclusively with other variables in L_i, $P^{\text{MVA-DG}}[d] = P^{\text{Gibbs}}[d]$. Thus, $\hat{\mathbf{z}}^{t+1} = \mathbf{z}^{t+1}$, which contradicts the hypothesis that $T^M \neq T^D$. □

Corollary 1. *The MVA framework with D-Gibbs as the global DCOP algorithm and Gibbs as the local search algorithm maintains the quality guarantees of both algorithms, that is, after* $\boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}$ *number of samples, the probability that the best solution found thus far* $\boldsymbol{x_N}$ *is in the top* α*-percentile is at least* $1-\epsilon$*. In other words,*

$$P_{Gibbs}\left(\boldsymbol{x_N} \in S_\alpha \mid \boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}\right) \geq 1 - \epsilon$$

Additionally, the quality of the solution found approaches optimal as the number of samples $\boldsymbol{N}$ *approaches infinity. In other words,*

$$\lim_{\epsilon \to 0} P_{Gibbs}\left(\boldsymbol{x_N} \in S_\alpha \mid \boldsymbol{N} = \frac{1}{\alpha \cdot \epsilon}\right) = 1$$

Corollary 1 is a direct consequence of Theorem 2 and Corollary 1 introduced by Nguyen, Yeoh, and Lau [84].

3.4 Related Work

To the best of our knowledge, the only algorithm able to deal with agent subproblems without the use of decomposition techniques is AdoptMVA [24], an extension of ADOPT [81]. The MVA decomposition presented here allows the integration of *any* global DCOP coordination algorithm and *any* local optimization procedure. As such, it subsumes AdoptMVA. Another line of work that solves subproblems in a centralized manner can be found in the work on *Partially Centralized (PC)* DCOP algorithms [100, 116]. The main difference with our approach is that the subproblems defined by the MVA decomposition are confined within the agents local variables, and therefore are privacy-preserving. In contrast, subproblems solved by PC algorithms are defined over variables that can be owned by different agents, which is undesirable in several application domains.

3.5 Experimental Evaluation

We evaluate our MVA decomposition with three global DCOP algorithms (AFB, DPOP, and D-Gibbs) and two local centralized solvers (DFBnB and Gibbs).In addition to the *lazy* version (*MVA-lazy*) described in this paper, where agents solve their local subproblems on demand during the resolution process, we also implemented an *eager* version (*MVA-eager*), where agents populate their complete MVA table in a pre-processing step. We compare them against the Compilation and the Decomposition pre-processing techniques on random graph and radar coordination instances. In our version of Compilation, agents retain exclusively the solutions of the local problem, whose search space is explored via DFBnB. All experiments are performed on an Intel i7 Quadcore 3.4GHz machine with 16GB of RAM. We report runtime measured using the simulated time [111] metric as well as the number of external agent-to-agent messages and internal agent messages. We impose a timeout of 600sec of simulated time and a memory limit of 2GB. Results report the average over 50 runs, and are statistically significant with p-values < 0.001.[1]

Random Graph Instances

We create an n-node network, whose local constraint graphs density p_1^l produces $\lfloor |L_i|(|L_i|-1)p_1^l \rfloor$ edges among the local variables of each agent a_i, and whose (global) density p_1^g produces $\lfloor b(b-1)p_1^g \rfloor$ edges among boundary non-local variables, where b is the total number of boundary variables of the problem.

Figures 3.6 – 3.11 show the results on these random graphs, where AFB and DPOP use DFBnB as local solver, while D-Gibbs uses Gibbs. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better). We conducted six experiments, in each of which we set as default parameters for the number of agents $|\mathbf{A}|$, the number of local variables per agent $|L_i|$, domain size of each variable $|D_i|$, the densities p_1^l, p_1^g and constraint tightness p_2 are respectively, 4, 6, 4, 0.6, 0.4, 0.4. For the first experiment, we vary the number of agents $|\mathbf{A}|$ (Figure 3.6). For the second experiment, we vary the number of local variables per agent $|L_i|$ (Figure 3.7). For the third experiment, vary the ratio $|B_i|/|L_i|$ (Figure 3.8). In this experiment, we build a subgraph for each agent with $p_1^l = 0.6$ and create as many inter-agent constraints as necessary to reach the desired ratio. For the forth experiment, we vary the constraint density of the global constraint graph p_1^g (Figure 3.9). For the fifth experiment, we vary the constraint density of the local constraint graphs p_1^l (Figure 3.10). Finally, for the sixth experiment, we vary the constraint tightness p_2 (Figure 3.11). In all the experiments, starting from the highest function arity, we transform each clique involving k variables to a k-ary function with costs randomly chosen from $[1, 1000]$. We make the following observations:

- Unlike Decomposition, MVA and Compilation do not need internal agent communication since agent subproblems are solved locally within each agent.
- The number of external messages required by each framework is similar for DPOP and D-Gibbs. The reason is that both DPOP and D-Gibbs external messages number is linear in the number of agents, and in the number of samples as well for D-Gibbs. Both of these factors are independent of the number of local variables.
- AFB on MVA requires up to one order of magnitude fewer external messages compared to Compilation, and several orders of magnitude fewer compared to Decomposition. The reason is that AFB agents broadcasts messages to request for cost estimates and announce complete solutions. These broadcasts occur more regularly with Decomposition and Compilation than with the MVA decomposition.
- The number of messages and runtimes of both MVA versions are similar to each other, indicating

[1] t-test performed with null hypothesis: MVA-lazy-decomposed algorithms are faster than non-MVA-decomposed ones.

that agents in both versions ultimately construct the entire MVA table.

- At increasing of the number of local variables, the solving time for both decomposition and compilation techniques increases exponentially, for both AFB and DPOP. In D-Gibbs, the compilation technique ran out of memory even for all instances with more than 4 local variables per agent. In the MVA framework the solving time increases roughly linearly with respect to the number of variables of the problem. This observation holds for each global algorithm tested.
- The runtimes of the algorithms on MVA tend to their runtimes with Decomposition as the ratio of boundary variables increases. When all variables are boundary variables ($^{|B_i|}/_{|L_i|} = 1$), AFB and D-Gibbs are faster on MVA than with Decomposition, as the agents can solve their local subproblems quicker with centralized algorithms; DPOP is slower on MVA due to overhead.
- The runtime of all algorithms on all decomposition techniques increases at the increasing of the local constraint graph p_1^l values, and at the increasing of the global constraint graph p_1^g values. DPOP on the Compilation decomposition fails to solve instances with p_1^g greater than 0.5 due to memory limitations.
- At increasing of the constraint tightness p_2 (bigger values correspond to more permissive constraints) MVA-eager becomes gradually faster than MVA-lazy on AFB, due to the overhead of the latter of building the MVA-tables, which eventually will be completely explored by both decompositions. For the other algorithms, the trends for the two MVA decompositions are similar.
- In general, all the algorithms are fastest on the MVA framework followed by with the Decomposition and Compilation techniques.

In all our experiments we, in addition to the above results, we also measured the number of concurrent constraint checks. Their value correlate to that of the simulated runtime with average value of 0.958 [2].

Radar Coordination Instances

This problem models a set of radars, which collect real-time data on the location and importance of atmospheric phenomena, and a set of controllers, which can operate on a subset of the radars [59]. Each phenomenon is characterized by size and weight (i.e., importance). Radars have limited sensing ranges, which determine their scanning regions. The goal is to find a radar configuration that maximizes the utility associated with the scanned phenomena. Controllers are modeled as agents whose variables they control are radars. The domain of each variable represents the scanning regions of a radar. The utilities (which, in our case, are modeled as costs, taking negative values) are functions involving all radars that may detect a given phenomenon.

In our experiments, radars are equally spaced onto a grid, and each controller coordinates 16 radars (arranged in a $4{\times}4$ grid). Radars have four possible scanning directions, and phenomena are randomly generated across the grid until the underling constraint graph results connected. Table 3.1 tabulates the results. The first two rows report the grid configurations and the number of agents. We omit the results for Compilation because it failed to solve any of the instances. We also omit results for D-Gibbs as it cannot handle hard constraints. Similarly to the random graph instances, the MVA-based algorithms are faster than Decomposition. Unlike random graph instances, AFB with MVA-lazy is up to one order of magnitude faster than MVA-eager. AFB can successfully prune portions of the search space using the hard constraints. As a result, AFB agents with MVA-lazy, in contrast to MVA-eager, do not need to

[2]Data obtained averaging the experiments at varying the number of agents, using Spearman correlation.

configuration # agents	8x4 2	8x8 4	12x8 6	16x8 8	20x8 10
MVA-lazy	11	213	3186	33212	42090
MVA-eager	732	10391	27549	77489	82637
Decomposition	213	108818	178431	timeout	timeout

AFB-DFBnB simulated time (ms)

MVA-lazy	844	30312	225761	478955	timeout
MVA-eager	823	30396	225538	477697	timeout
Decomposition	61978	timeout	timeout	timeout	timeout

DPOP-DFBnB simulated time (ms)

Table 3.1: Radar Coordination Instances.

construct the entire MVA table. DPOP with both MVA-lazy and MVA-eager have similar runtimes, as DPOP does not perform any pruning being based on dynamic programming.

3.6 Summary

This chapter introduced the MVA decomposition for DCOPs with multi-variable agents. This decomposition defines a clear separation between the distributed agent coordination and the centralized agent subproblem resolution, while preserving agent privacy. This separation allows the use of efficient centralized solvers to solve agent subproblems as well as the use, for different agents, of potentially different solvers designed to exploit domain-specific properties. Experimental results show that the use of MVA speeds up several DCOP algorithms, by up to several orders of magnitude, and reduces their communication requirements with respect to existing techniques. These results experimentally validate the hypothesis that exploiting latent structure embedded into the general DCOP model, can speed up the resolution process.

An interesting property of our MVA decomposition is that it naturally enables the DCOP resolution process to the use of hierarchical parallel solutions. Such property is motivated by the observation that the search for the best local solution for each row of the MVA_TABLE is independent of the search for another row and, as such, they can be performed in parallel. Such hierarchical parallel model could to further speed up the local optimization process, thus reducing the overall DCOP solving time. This observation finds a natural fit for SIMT processing. We have exploited this property of the MVA decomposition to speed up the resolution approach of a particular class of sampling algorithms, which we will detail in section 5.3.

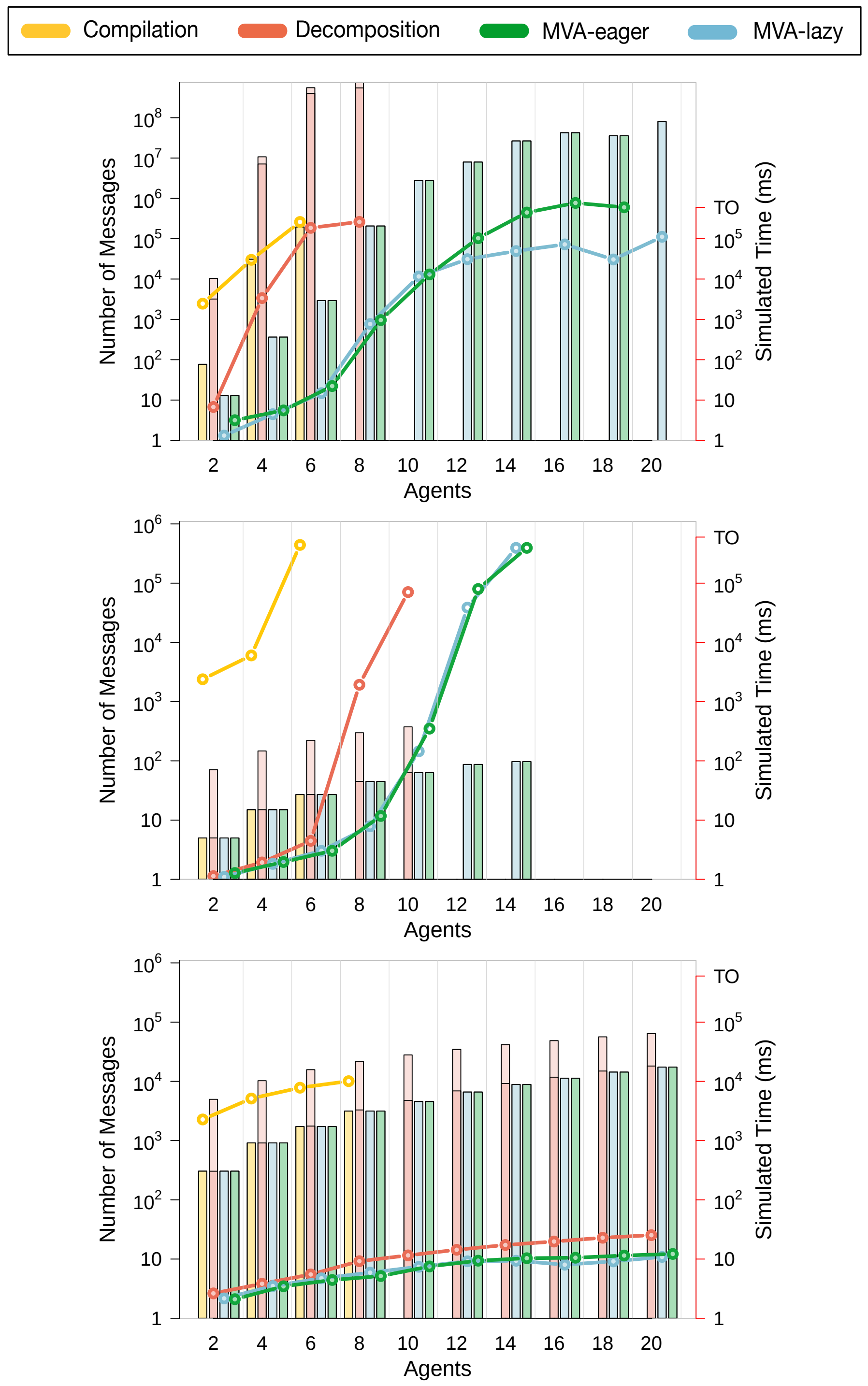


Figure 3.6: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the number of agents $\mathbf{A}$. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

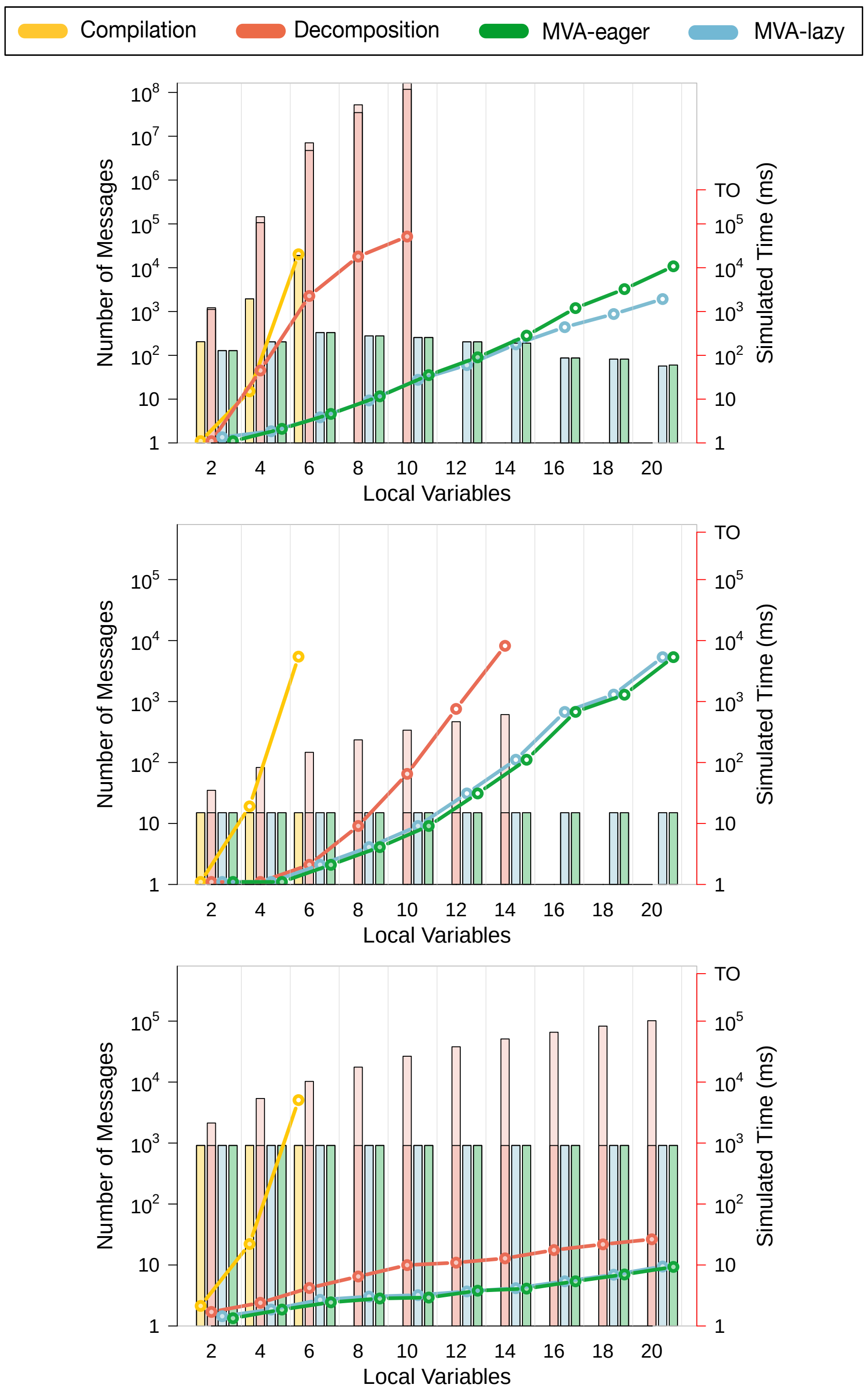


Figure 3.7: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the number of local variables L_i. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

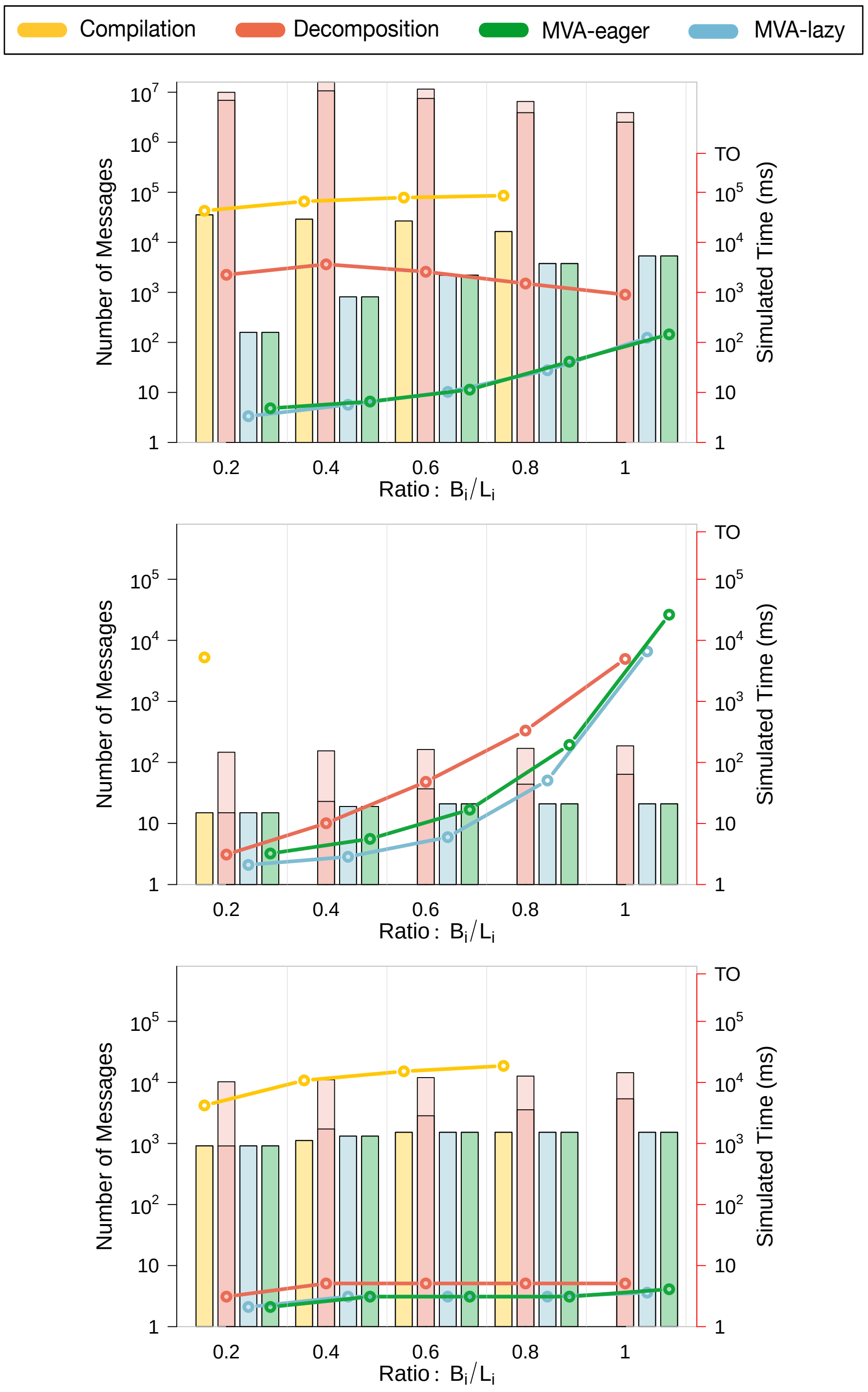


Figure 3.8: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the ratio $|B_i|/|L_i|$. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

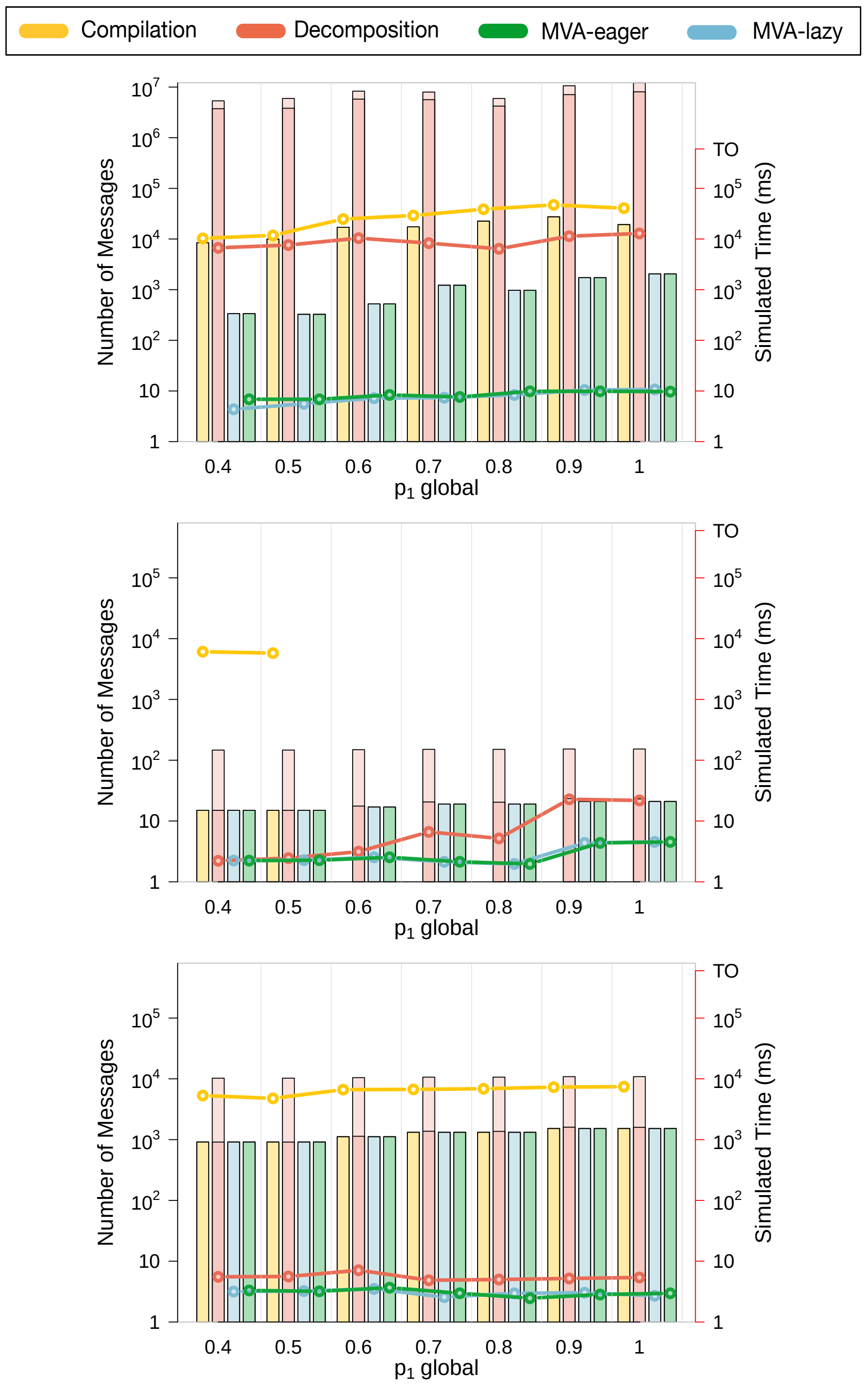


Figure 3.9: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the global constraint graph density p_1^g. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

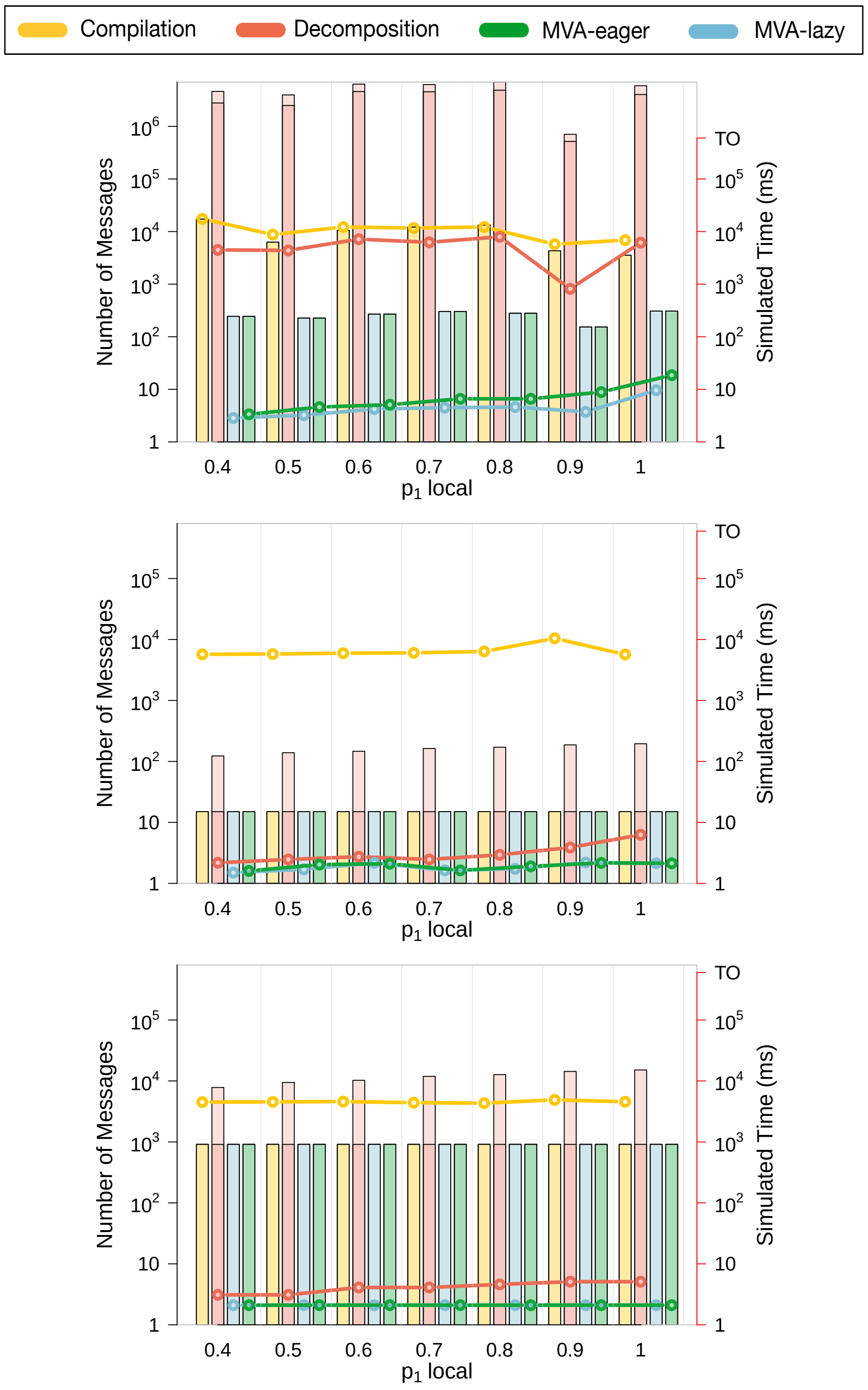


Figure 3.10: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the local constraint graph density p_1^l. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

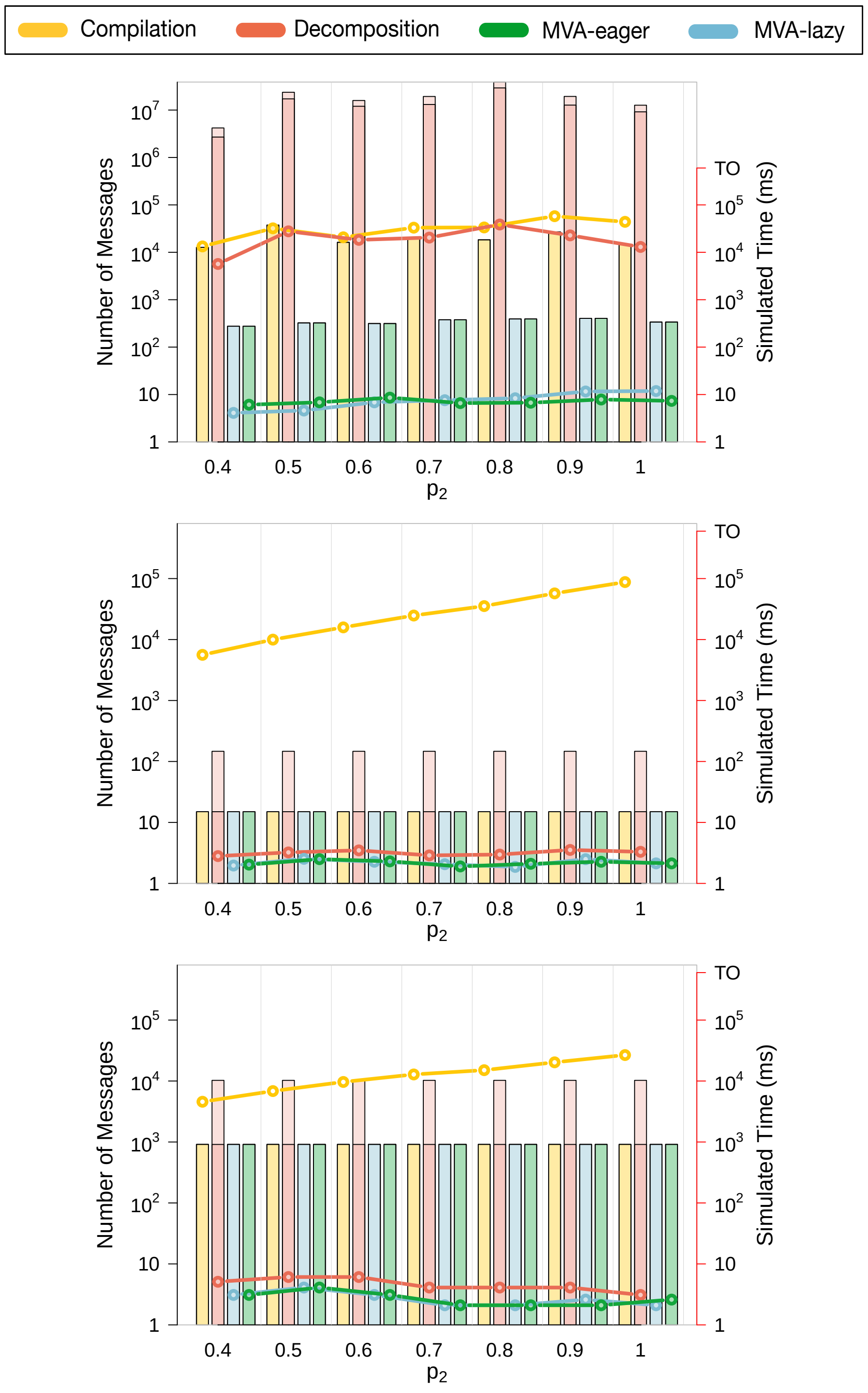


Figure 3.11: Random Graph instances for AFB-DFBnB (top), DPOP-DFBnB (center), and D-Gibbs-Gibbs (bottom), at varying of the constraint tightness p_2. Dark (light) bars indicate the number of external (internal) agent-to-agent messages, and lines indicate runtime, all in logarithmic scale (the smaller, the better).

4

Exploiting the Structure of DCOPs during Problem Solving

This chapter introduces *Branch Consistency* (BrC), and *Distributed Large Neighboring Search* (D-LNS), two DCOP solving strategies which adapt centralized reasoning techniques to speed up the DCOP resolution by exploiting the structure of DCOPs in orthogonal ways. The first, is a type of consistency that applies to paths in pseudo-trees, and it is aimed to prune the search space and to reduce the size of the messages exchanged among agents by actively exploiting the hard constraints of the problem. The second, is a local search framework for DCOPs which builds on the strengths of centralized *Large Neighboring Search* (LNS) [55], a centralized meta-heuristic that iteratively explores complex neighborhoods of the search space to find better candidate solutions. D-LNS can exploit problem structure from domain-dependent knowledge, and it inherently uses insights from the *CP* techniques to take advantage on the presence of hard constraints, to further enhance runtime and solution quality of the problem being solved. Our experimental results show that BrC enforces a more effective pruning than those based on domain-consistency, guaranteeing optimality, and leading enhanced efficiency and scalability. Furthermore, D-LNS based algorithms converge faster to better solutions, compared to other incomplete algorithms, and provide tighter solution quality bounds. Therefore, these results validate the hypothesis that centralized reasoning can be adapted to exploit the structure of DCOPs during problem solving to enhance the DCOP solving efficiency.

This chapter is structured as follows: We first describe the motivations for our two solving strategies in section 4.1, we thus detail the description of BrC in section 4.2, and therein introduce BrC-DPOP, an algorithm that exploits BrC to reduce the size of the messages exchanged by the agents during problem solving. section 4.3 introduces the D-LNS framework as well as two D-LNS based algorithms characterized by the ability to exploit domain dependent structure, low network usage, and low computational complexity per agent. In each of the latter two sections we report the description of the algorithms, a theoretical analysis on the relevant properties exposed by the two solving strategies, and the experimental results. Finally, we conclude with our summary, presented in section 4.4.

4.1 Motivations

4.1.1 Hard Constraints

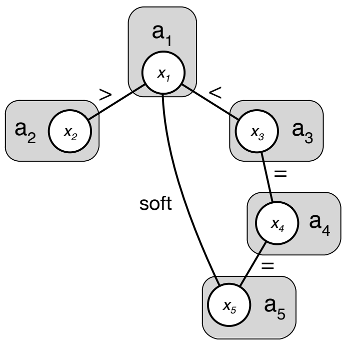
Many real-world distributed constraint optimization models require the use of hard constraints to avoid considering infeasible solutions (see, e.g., `http://www.csplib.org` for an extensive list of problem domains). Several works from the DCOP community have recognized the importance of hard constraints to reduce the size of the search space and/or reduce the message size exchanged by the agents [50, 63]. However, they are limited in exploiting relational information expressed in form of tables and/or associated to the form of domain consistency, which may result in arbitrarily poor pruning. Thus, we hypothesize that by using insights from the centralized constraint reasoning community, DCOP agents can actively exploit hard constraints in a distributed fashion to enforce a more effective pruning, increasing the efficiency of the network load and reducing the time of the resolution process. To verify such hypothesis we propose a novel type of consistency, called *Branch Consistency* (BrC), that applies to paths in pseudo-trees [35]. Such form of consistency enforces a more effective pruning than those based on domain-consistency, guaranteeing optimality, and leading enhanced efficiency and scalability. We have applied such form of consistency to reduce the space explored by DPOP agents, and consequentially reduce the size of the messages exchanged, resulting in a new algorithm, called *BrC-DPOP* and introduced in section 4.2. Importantly, *BrC-DPOP* agents can effectively exploit the information encoded in the hard constraints of the problem, reducing their message size, and without incurring in any additional privacy loss.

4.1.2 Large, Complex Problems

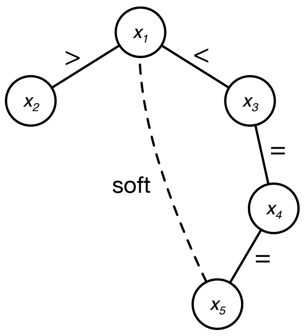
In many cases, the coordination protocols required for the complete resolution of DCOPs demand a vast amount of resources and/or communication, making them infeasible to solve real-world complex problems. Since finding optimal DCOP solutions is NP-hard, incomplete algorithms are often necessary to solve large interesting problems. Desirable properties of good incomplete DCOP algorithms include, *(i)* to quickly converge to good local minima; *(ii)* to be anytime; and *(iii)* to provide guarantees on solutions quality. Being in a distributed setting, they are also required to exhibit low resources usage (i.e., network load and bandwidth). We hypothesize that such properties can be obtained by capitalizing on the strengths of a local search strategy widely adopted within the centralized constraint reasoning community: the *Large Neighboring Search* (LNS) [55]. To validate such hypothesis we propose the *Distributed Large Neighboring Search* (D-LNS) framework [32], which solves DCOPs by building on the strengths of centralized LNS. The resulting framework has several qualities: It provides quality guarantees by refining both upper and lower bounds of the solution found during the iterative process; It is anytime; and it inherently uses insights from the *CP* techniques to take advantage on the presence of hard constraints. To illustrate the generality of the proposed framework we introduce two novel distributed search algorithms in section 4.3, built within D-LNS, characterized by the ability to exploit problem structure, low network usage, and low computational complexity per agent.

4.2 Branch Consistency to Exploit Hard Constraints

This section introduces *Branch Consistency* (BrC), a new form of local consistency that applies to variables ordered in a pseudo-tree. BrC can be viewed as a weaker version of path consistency [82] tailored to variables within the same branch of the pseudo-tree, and where each agent can only communicate



(a) Constraint Graph



(b) Pseudo-tree

x_1	x_5	Utilities
0	0	20
0	1	8
0	2	10
0	3	3
	...	
3	3	2

(c) Constraint Table

Figure 4.1: Example DCOP

exclusively with neighboring agents, and thus, BrC is suitable to be applied to DCOPs. In addition we propose a novel variant of DPOP, called *Branch-Consistency DPOP* (BrC-DPOP), that takes advantage of the hard constraints present in the problem to prune the search space by enforcing Branch Consistency. The effect of enforcing this consistency in DPOP is that of generating smaller UTIL tables, and thus to effectively reduce the size of the messages exchanged among agents, up to several order of magnitude, as shown in our experimental evaluation in section 4.2.5

Through the section, we will use the example DCOP shown in Figure 4.1, to illustrate the effect of enforcing BrC in the size of messages exchanged by BrC-DPOP agents. Figure 4.1(a) shows the constraint graph of a simple DCOP with five agents, a_i, with $i = 1, \dots, 5$, each owning exactly one variable x_i. The domain of each variable is the set $\{0, 1, 2, 3\}$. Figure 4.1(b) shows one possible pseudo-tree for the problem, where the agent a_1 has one pseudo-child, a_5 (the dotted line is a backedge). Figure 4.1(c) describes few value combinations of the utility function associated with the constraint f_{15}.

4.2.1 Notation and Definitions

We now introduce the concept of *Branch Consistency* and some related notions adopted by our proposed algorithm. We restrict our attention to unary and binary utility functions and refer to unary constraints as f_{ii} and binary constraints as f_{ij} to denote the fact that their scope is $\{x_i\} \subseteq \mathbf{X}$ and $\{x_i, x_j\} \subseteq \mathbf{X}$, respectively. We assume at most one constraint between each pair of variables, thus making the order of variables in the scope of a constraint irrelevant. To simplify notation, we also assume that each agent controls exactly one variable, and thus, use the terms "variable" and "agent" interchangeably.

Definition 8 (Consistency Graph)**.** *The* consistency graph *of a DCOP* $P = (\mathbf{A}, \mathbf{X}, \mathbf{D}, \mathbf{F}, \alpha)$ *is* $\ddot{G}_P = (\mathbf{V}, \mathbf{E})$ *where* $\mathbf{V} = \{(i,k) \,|\, x_i \in \mathbf{X}, k \in D_i\}$ *and* $\mathbf{E} = \{\langle (i,r), (j,c) \rangle \,|\, r \in D_i, c \in D_j, f_{ij} \in \mathbf{F}, (r,c) \in f_{ij}\}$.

The consistency graph of a DCOP is useful to visualize the values for pairs of variables which are consistent given the constraint of the problem.

Example 1. *Consider the DCOP of Figure 4.1 with domains for the variables* x_i, $(i = 1, \dots, 5)$ *being*

$D_i = \{0,1\}$. *Then the consistency graph* $\ddot{G} = (V, E)$ *with* $V = \{(i,j)\}$ *with* $i = 1, \ldots, 5$, *and* $j = \{0,1\}$, *and* $E = \{\langle(1,0),(2,1)\rangle, \langle(1,1),(3,0)\rangle, \langle(3,0),(4,0)\rangle, \langle(3,1),(4,1)\rangle, \langle(4,0),(5,0)\rangle, \langle(4,1),(5,1)\rangle\}$

Definition 9 (Linear Ordering). *Given a pseudo-tree* T_P *associated with a DCOP* P, *we define a* linear ordering $\prec$ *on its variables:* $x_i \prec x_j$ *if and only if* $x_j \in P_{a_i}$. *Similarly,* $x_i \succ x_j$ *if and only if* $x_j \in C_{a_i}$. *We denote with* $\preceq$ *(and* $\succeq$*) the reflexive closure of* $\prec$ *(and* $\succ$*), and with* $\overset{*}{\prec}$ *(and* $\overset{*}{\succ}$, $\overset{*}{\preceq}$, $\overset{*}{\succeq}$*) the transitive closure of* $\prec$ *(and* $\succ$, $\preceq$, $\succeq$ *).*

A linear ordering defines a precedence relation over the variables of a DCOP and it is useful to determine paths from ancestors variables to successors variables, and vice versa, traversing exclusively tree edges.

Example 2. *Consider the pseudo-tree of the DCOP in Figure 4.1(b). It can be observed that* $x_5 \prec x_4$, $x_4 \prec x_3$, *and* $x_5 \overset{*}{\preceq} x_3$, *however* $x_2 \overset{*}{\npreceq} x_i$, *with* $i = 3, 4, 5$.

Definition 10 (Branch Consistency (for pair of values)). *A pair of values* $(r, c) \in D_i \times D_j$ *of two variables* x_i, x_j *that share a constraint* f_{ij} *is* branch consistent (BrC) *if and only if for any sequence of variables* $(x_i = x_{k_1}, \ldots, x_{k_m} = x_j)$, *such that* $f_{k_p k_q} \in \mathbf{F}$, *where* $p \leq q \leq p+1$, *and* $x_{k_1} \preceq \cdots \preceq x_{k_m}$, *there exists a tuple of values* $(r = v_{k_1}, \ldots, v_{k_m} = c)$ *such that* $v_{k_q} \in D_{k_q}$ *and* $(v_{k_p}, v_{k_q}) \in f_{k_p k_q}$, *for each* $1 \leq q \leq m$ *and* $p \leq q \leq p+1$.

Example 3. *Consider the DCOP of Figure 4.1 with domains for the variables* x_3, x_4, *and* x_5 *being* $D_3 = \{0,1\}, D_4 = \{1,2\}, D_5 = 0,1,2$. *The pair of values* $(1,1)$ *for the variables* x_3 *and* x_5 *is BrC, while the pair of values* $(0,2)$ *is not BrC because neither* $\langle(3,0),(4,1)\rangle$ *nor* $\langle(3,0),(4,2)\rangle$ *can be extended to the value of* $(5,2)$ *satisfying both* f_{34} *and* f_{45}.

Definition 11 (Branch Consistency). *A DCOP is* branch consistent (BrC) *if and only if for any pair of variables* (x_i, x_j) *with* $x_i \preceq x_j$ *and any* $(u, v) \in f_{ij}$, (u, v) *is branch consistent.*

Definition 12 (Value Reachability Matrix). *Given a DCOP, the* Value Reachability Matrix (VRM) M_{ij} *of two variables* x_i *and* x_j *of* $\mathbf{X}$, *with* $x_i \overset{*}{\preceq} x_j$, *is a binary matrix of size* $D_i \times D_j$, *where* $M_{ij}[r, c] = 1$ *if and only if there exists at least one sequence of variables* $(x_i = x_{k_1}, \ldots, x_{k_m} = x_j)$, *such that* $x_{k_1} \preceq \cdots \preceq x_{k_m}$, *and a tuple of values* $(r = v_{k_1}, v_{k_2}, \ldots, v_{k_m} = c)$ *such that* $v_{k_p} \in D_{k_p}$ *and* $(v_{k_p}, v_{k_q}) \in f_{k_p k_q}$, *for each* $1 \leq q \leq m$ *and* $p \leq q \leq p+1$.

A VRM M_{ij} between variables x_i and x_j represents the pair of values that can be extended along the linear ordered paths between x_i and x_j.

Example 4. *Consider the DCOP of Example 3. The VRM* M_{35} *of variables* x_3 *and* x_5 *is:*

$$M_{35} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Proposition 1. *For each variable* x_i, x_j, *and* x_k, *the* regular product *of two VRMs* M_{ik} *and* M_{kj} *is a VRM* $M_{ij} = M_{ik} \times M_{kj}$, *where each entry* (r, c) *of* M_{ij} *is given by*

$$M_{ij}[r, c] = \min \left\{ 1, \sum_{l=1}^{|D_k|} M_{ik}[r, l] \cdot M_{kj}[l, c] \right\}$$

The regular product of two VRMs is defined as the regular product of two *general* matrices, where all the resulting entries greater or equal than 1 are reduced to 1.

Proposition 2. *For each variable x_i and x_j, the* entrywise product *of two VRMs M_{ij} and $\hat{M}_{ij}$ is a VRM $M'_{ij} = M_{ij} \circ \hat{M}_{ij}$, where each entry (r, c) of M'_{ij} is given by*

$$M'_{ik}[r, c] = M_{ij}[r, c] \cdot \hat{M}_{ij}[r, c]$$

The entrywise product of two VRMs is defined as the entrywise product of two *general* matrices.

Definition 13 (Valid pair)**.** *Given a VRM M_{ij}, a pair of values (r,c) is a* valid *pair if and only if $M_{ij}[r, c] = 1$.*

Proposition 3. *If $f_{ij} \in$ **F**, then M_{ij} is* branch consistent (BrC) *if and only if all its valid pairs are branch consistent. If $f_{ij} \notin$ **F**, then M_{ij} is* branch consistent *if and only if it is a regular product of branch consistent VRMs.*

4.2.2 BrC-DPOP

We now describe BrC-DPOP, an algorithm which builds on DPOP, and that makes use of the concepts introduced in the previous section to prune unfeasible portions of the search space, exploiting the hard constraints of the problem.

High-Level Algorithm Description

We first illustrate the high-level structure of BrC-DPOP on the example DCOP shown in Figure 4.1. BrC-DPOP consists of the following phases:

- **Pseudo-tree Generation Phase**: Similarly to the Pseudo-tree Generation Phase of DPOP, in this phase the agents coordinate the construction of a pseudo-tree.
- **Path Construction Phase**: In this phase, each agent builds the VRMs associated with the constraints involving its variables along with the structures describing the paths between pseudo-parents and pseudo-children. Figure 4.2(a) shows the VRMs (in a consistency graph representation); we do not show the soft constraint between variables x_1 and x_5 as it allows every value combination of the two variables.
- **Arc Consistency Enforcement Phase**: In this phase, the agents enforce arc consistency in a distributed manner. At the end of this phase, each agent has the updated VRMs shown in Figure 4.2(b). Arc consistency causes the removal of exactly two values from the domain of each variable of the DCOP: values 0 and 3 from D_1, 0 and 1 from D_2, and 2 and 3 from D_3, D_4, and D_5.
- **Branch Consistency Enforcement Phase**: In this phase, the agents enforce branch consistency in a distributed manner. In our example, branch consistency needs to be enforced for the pairs of values of variables x_1 and x_5 only. The values for all other pairs of variables are already branch consistent. Agent a_1 starts this process by sending a message containing VRM M_{11} to its child a_3 (since a_5 is in the subtree rooted at a_3). Once agent a_3 receives the message, it computes the VRM M_{31} by multiplying its VRM M_{31} with the VRM M_{11} just received, and sends a message containing this VRM to its child a_4. Agent a_4 repeats this process by multiplying its VRM M_{43} with the VRM

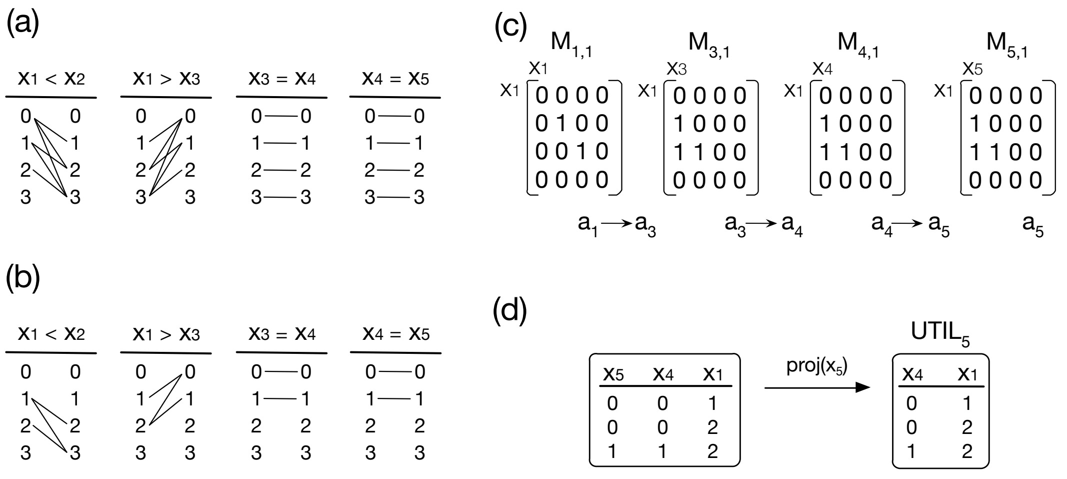


Figure 4.2: BrC-DPOP Example Trace

M_{31}, resulting in VRM M_{41}, which it sends to its child a_5. This process repeats until agent a_5 computes the VRM M_{51}, after which it knows its set of reachable values in x_5 for each value in x_1. Figure 4.2(c) shows the VRMs.

- **UTIL and VALUE Propagation Phases**: This phase is identical to the corresponding UTIL and VALUE propagation phases of DPOP, except that each agent constructs a UTIL table that contains utilities for each combination of *unpruned* values of variables in its VRMs. In our example, agent a_5 is able to project out its variable x_5 and construct its UTIL table, shown in Figure 4.2(d). Note that the UTIL table consists of only 3 utilities, both before and after projection. In contrast, DPOP's UTIL table consists of $4^3 = 64$ utilities before projection and $4^2 = 16$ utilities after projection.

Messages and Data Structures

During the execution of BrC-DPOP, each agent a_i maintains the following data structures, where the first three are used in the arc consistency phase and the last two are used in the branch consistency phase.

- The set of hard constraints $\hat{\mathbf{H}}_i = \{f_{ij} \in \mathbf{H}_i \mid a_i \overset{*}{\preceq} a_j\}$ to check for consistency.
- The set of VRMs $\hat{\mathbf{M}}_i = \{\hat{M}_{ij} \mid f_{ij} \in \mathbf{F}, a_j \overset{*}{\preceq} a_i\}$, which includes the VRMs for each parent and pseudo-parent a_j.
- The flag *fixed*$_i$ for each agent a_i, which is initialized to true. It indicates weather agent a_i has reached a fixed point in the arc consistency phase.
- The set of VRMs $\mathbf{M}_i = \{M_{ij} \mid a_j \in sep(a_i)\}$, which includes the VRMs for each agent a_j in the separator set of the agent a_i.
- The set of paths $Paths_{a_i} = \{(a_s \overset{a_j}{\rightsquigarrow} a_d) \mid a_j \in C_{a_i},\ a_s \overset{*}{\succeq} a_i \succ a_j \overset{*}{\succeq} a_d\}$, which the agent uses to send updated VRMs in the branch consistency phase. Each path $(a_s \overset{a_j}{\rightsquigarrow} a_d)$ indicates that there is

Algorithm 2: BRC-DPOP

```
1 Pseudo-tree-Generation-Phase( )
2 Path-Construction-Phase( )
3 AC-Propagation-Phase( )
4 BrC-Propagation-Phase( )
5 UTIL-and-VALUE-Phases( )
```

a branch in the pseudo-tree from a_s to a_d that traverses a_i and its child a_j. This data is needed by agent a_i to know which child it should send its updated VRM to, if the VRM originated from agent a_s. For instance, in our example trace, agent a_1 knows to send its VRM to its child a_3 and not a_2. To preserve privacy, the information about the destination agent a_d can be omitted from each path. Each agent thus maintains only $(a_s \overset{a_j}{\rightsquigarrow} ?)$, which is sufficient to ensure correctness.

In addition to the UTIL and VALUE messages used in the UTIL and VALUE propagation phases, each agent a_i uses the following types of messages, where the first two are used in the arc consistency phase, while the last two in the branch consistency phase:[1]

- $[\text{AC}]_i^{\uparrow}(D'_j, \textit{fixed}_i)$, which is sent from an agent a_i to an agent $a_j \overset{*}{\succ} a_i$ such that $f_{ij} \in \mathbf{H}_i$. It contains a copy of the domain of the variable x_j, D'_j, updated with the changes caused by the propagation of the constraints in $\hat{\mathbf{H}}_i$, and a flag, *fixed*$_i$, which denotes whether changes have occurred in the domain of some variable in the subtree rooted at a_i during the last iteration of the $AC^{\uparrow}$ messages.
- $[\text{AC}]_i^{\downarrow}(D_i)$, which is sent from an agent a_i to the agents $a_j \overset{*}{\prec} a_i$ such that $f_{ij} \in \mathbf{H}_i$. It contains a copy of the domain of the variable x_i, D_i, updated with the changes caused by the propagation of the constraints in $\hat{\mathbf{H}}_i$.
- $[\text{PATH}]_i^{\uparrow}(a_s)$, which is sent from an agent a_i to its parent P_{a_i} to inform it that it is part of a tree path in the pseudo-tree between agents a_s and some pseudo-child of a_s.
- $[\text{BrC}]_i^{\downarrow}(M_{is})$, which is used to determine the branch consistent value pairs of x_s and x_i.

Algorithm Description

Algorithm 2 shows the pseudo-code of BrC-DPOP. It can be visualized as a process composed of 5 phases:

- **Phase 1 - Pseudo-tree Generation Phase:** This phase is identical to that of DPOP, where a pseudo-tree is generated (line 1).
- **Phase 2 - Path Construction Phase:** The phase is used to construct the direct paths from each agent to its parent and pseudo-parents. At the start of this phase (line 2), each agent, starting from the leaves of the pseudo-tree, recursively populates its $Paths_{a_i}$ as follows: It saves a path information $(a_p \overset{NULL}{\rightsquigarrow} ?)$ for each of its pseudo-parents a_p (lines 6-7) and sends a $[\text{PATH}]_i^{\uparrow}(a_p)$ message to its parent. When the parent a_i receives a $[\text{PATH}]_c^{\uparrow}$ message from each of its child a_c, it stores the path information in the message in its $Paths_{a_i}$ data structure (lines 9-11). For each path in $Paths_{a_i}$, if it is not the destination

[1]We use the superscript $\uparrow$ to denote the messages being propagated from the leaves of the pseudo-tree to the root, and $\downarrow$ to denote the ones propagated from the root to the leaves.

Procedure Path-Construction-Phase( )

```
6  foreach $a_p \in PP_{a_i}$ do
7      $Paths_{a_i} \leftarrow Paths_{a_i} \cup (a_p \overset{NULL}{\rightsquigarrow} ?)$
8  if $C_{a_i} \neq \emptyset$ then
9      while not received all $[\texttt{PATH}]_c^{\uparrow}(\cdot)$ from each $a_c \in C_{a_i}$ do
10         if receive $[\texttt{PATH}]_c^{\uparrow}(a_s)$ from $a_c \in C_{a_i}$ then
11             $Paths_{a_i} \leftarrow Paths_{a_i} \cup (a_s \overset{a_c}{\rightsquigarrow} ?)$
12 foreach $a_s \neq a_i$ such that $(a_s \overset{a_c}{\rightsquigarrow} ?) \in Paths_{a_i}$ do
13     Send $[\texttt{PATH}]_i^{\uparrow}(a_s)$ to $P_{a_i}$
14 if $[\texttt{PATH}]_i^{\uparrow}(\cdot)$ has not been sent to $P_{a_i}$ then
15     Send $[\texttt{PATH}]_i^{\uparrow}(NULL)$ to $P_{a_i}$
```

agent, then it sends a $[\texttt{PATH}]_j^{\uparrow}$ message that contains that path to its parent (lines 12-13). If it does not send a $[\texttt{PATH}]_j^{\uparrow}$ message to its parent, then it sends an empty $[\texttt{PATH}]_j^{\uparrow}$ message (lines 14-15). These path information will be used in the branch consistency propagation phase later. When the root processes and stores the path information from each of its children, it ends this phase and starts the next AC propagation phase.

- **Phase 3 - Arc Consistency (AC) Propagation Phase:** In this phase, the agents enforce arc consistency in a distributed manner by interleaving the direction of the AC message flows: from the leaves to the root (lines 18-24) and from the root to the leaves (lines 25-34), until a fixed point is detected at the root (line 35).

  In the first part of this phase (lines 18-24), each agent, starting from the leaves up to the root, recursively enforces the consistency of its hard constraints in $\hat{\mathbf{H}}_i$ (line 22) via the *enforceAC* procedure, which we implemented using the AC-2001 algorithm [6]. In this process, the agent also updates the VRMs $\hat{\mathbf{M}}_i$ associated with all its constraints $f_{ij} \in \hat{\mathbf{H}}_i$ and its domain D_i to prune all unsupported values. If any of its values are pruned, indicating that it has not reached a fixed point, it sets its *fixed*$_i$ flag to false (line 23). It then sends an $[\texttt{AC}]_i^{\uparrow}$ message to each of its parent and pseudo-parent a_j, which contains its *fixed*$_i$ flag as well as a copy of their domains D'_j[2] to notify them about which unsupported values were pruned (line 24). The domain of each agent is updated before enforcing the arc consistency, as soon as it receives all the $[\texttt{AC}]_i^{\uparrow}$ messages from each of its children and pseudo-children (lines 20-21).

  Once the root enforces the consistency of its hard constraints, it checks if it has reached a fixed point (line 28). If it has not, then it starts the next part of this phase, which is similar to the previous one except for the direction of the recursion and the AC message flow (lines 29-34). This phase is carried from the root down to the leaves of the pseudo-tree, and it ends when all the leaves have enforced the consistency of their hard constraints. Then the procedure repeats the first part where the recursion and the AC message flow starts from the leaves again and continues up to the root. This process repeats until a fixed point is reached at the root (line 35), which ends this phase, and starts the next BrC propagation phase.

[2]In the pseudo-code, we use the notation $\hat{M}_{ij|j}$ to indicate D'_j.

Procedure AC-Propagation-Phase( )

```
16 $iteration \leftarrow 0$
17 repeat
18     if $C_{a_i} \neq \emptyset$ then
19         Wait until received $[\texttt{AC}]_c^{\uparrow}(D'_i, \textit{fixed}_c)$ from each $a_c \in C_{a_i} \cup PC_{a_i}$ in this iteration
20     foreach $[\texttt{AC}]_c^{\uparrow}(D'_i, \textit{fixed}_c)$ received do
21         $D_i \leftarrow D_i \cap D'_i$
22     $\langle \hat{\mathbf{M}}_i, D_i \rangle \leftarrow \textit{enforceAC}(\hat{\mathbf{H}}_i, \hat{\mathbf{M}}_i, D_i)$
23     $\textit{fixed}_i \leftarrow \neg changed(D_i) \wedge \bigwedge_{a_c \in C_{a_i}} \textit{fixed}_c$
24     Send $[\texttt{AC}]_i^{\uparrow}(\hat{M}_{ij|j}, \textit{fixed}_i)$ to each $a_j \in P_{a_i} \cup PP_{a_i}$
25     if $P_{a_i} \neq \textit{NULL}$ then
26         Wait until received $[\texttt{AC}]_p^{\downarrow}(D_p)$ from each $a_p \in P_{a_i} \cup PP_{a_i}$ in this iteration or received
           $[\texttt{BrC}]_p^{\downarrow}(\cdot)$ from parent $a_p$
27         if received $[\texttt{BrC}]_p^{\downarrow}(\cdot)$ from parent $a_p$ then break
28     if $\neg \textit{fixed}_i$ then
29         foreach $[\texttt{AC}]_p^{\downarrow}(D_p)$ received do
30             update $\hat{M}_{ip}$ with $D_p$
31         if $P_{a_i} \neq \textit{NULL}$ then
32             $\langle \hat{\mathbf{M}}_i, D_i \rangle \leftarrow \textit{enforceAC}(\hat{\mathbf{H}}_i, \hat{\mathbf{M}}_i, D_i)$
33         Send $[\texttt{AC}]_i^{\downarrow}(D_i)$ to each $a_c \in C_{a_i} \cup PC_{a_i}$
34         $iteration \leftarrow iteration + 1$
35 until $P_{a_i} = \textit{NULL}$ and $\textit{fixed}_i$;
```

- **Phase 4 - Branch Consistency (BrC) Propagation Phase:** In this phase, the agents enforce branch consistency in a distributed manner, that is, every pair of values of an agent and its pseudo-parents are mutually reachable throughout every tree path connecting them in the pseudo-tree.

  At the start of this phase, each agent, starting from the root down to the leaves, recursively enforces branch consistency for all tree paths from the root to that agent and sends a $[\texttt{BrC}]_i^{\downarrow}$ message to each of its children. This message includes the VRM for each path through that child. Once an agent a_i receives all the VRM messages from its parent (lines 36-37), for each path that goes through it (line 38), it creates a new VRM M_{is}. If it is the start of the path, then it sets its VRM $\hat{M}_{ii}$ (line 39-40), which is arc consistent, as the new VRM M_{is}. Otherwise, it performs the regular product of its VRM $\hat{M}_{ip}$ for the constraint between itself and its parent a_p and the VRM received from the parent M_{ps} and sets it to M_{is} (line 42). Then, to ensure that the VRM M_{is} is branch consistent, it performs the *entrywise product* with the VRM $\hat{M}_{is}$ of its pseudo-parent a_s (line 43). If the agent is the destination of the path, then it will use the resulting VRM in the construction of the UTIL messages in the UTIL phase. Otherwise, it will send the VRM to its child agent that is in that path in a $[\texttt{BrC}]_i^{\downarrow}$ message (lines 44-45). Finally, it will send an empty $[\texttt{BrC}]_i^{\downarrow}$ to all remaining child agents to ensure that the propagation reaches all the leaves (lines 46-47).

- **Phase 5 - DPOP's UTIL and VALUE Phases:** This phase is identical to the corresponding UTIL

Procedure BrC-Propagation-Phase( )

```
36 if $P_{a_i} \neq NULL$ then
37     Wait until received a $[\texttt{BrC}]^{\downarrow}_{p}(M_{ps})$ for each path $(a_s \overset{a_c}{\rightsquigarrow} ?) \in Paths_{a_i}$ from parent $a_p$
38 foreach $(a_s \overset{a_c}{\rightsquigarrow} ?) \in Paths_{a_i}$ do
39     if $a_s = a_i$ then
40         $M_{is} \leftarrow \hat{M}_{ii}$
41     else
42         $M_{is} \leftarrow \hat{M}_{ip} \times M_{ps}$
43     $M_{is} \leftarrow \hat{M}_{is} \circ M_{is}$
44     if $a_c \neq NULL$ then
45         Send $[\texttt{BrC}]^{\downarrow}_{i}(M_{is})$ to $a_c$
46 foreach $a_c \in C_{a_i}$ that has not been sent a $[\texttt{BrC}]^{\downarrow}_{i}$ message do
47     Send $[\texttt{BrC}]^{\downarrow}_{i}(NULL)$ to $a_c$
```

and VALUE propagation phases of DPOP, except that each agent constructs a UTIL table that contains utilities for each combination of *unpruned* values of variables in its VRMs.

4.2.3 Theoretical Analysis

In this section, we provide a complexity analysis for the network load and the messages size for the AC and BrC propagation phases. We prove the correctness and completeness of BrC-DPOP, and report the network load complexity for the UTIL and VALUE phases of BrC-DPOP, as well as its agents' memory requirement. In the following, we use n, k, and d to denote $|\mathbf{X}| = |\mathbf{A}|$, $|\mathbf{F}|$, and $\max_{x_i \in \mathbf{X}} |D_i|$, respectively.

Theorem 5. *The AC propagation phase requires $O(ndk)$ messages, each of size $O(d)$.*

Proof. In the worst case, each AC iteration removes exactly one value from one domain. Thus, there are only $O(nd)$ iterations, as there are only $O(nd)$ values among all variables. In each iteration, each agent sends exactly one $[\texttt{AC}]^{\uparrow}$ message to each parent and pseudo-parent and one $[\texttt{AC}]^{\downarrow}$ message to each child and pseudo-child. Thus, there are at most $O(k)$ messages sent in each iteration. Each message contains at most the full domain of a variable and the fixed flag, which is $O(d)$. □

Theorem 6. *The BrC propagation phase requires $O(k)$ messages, each of size $O(d^2)$.*

Proof. In the BrC propagation phase, each agent sends exactly one $[\texttt{BrC}]^{\downarrow}$ message to each child, and the phase ends after all the leaves in the pseudo-tree receives a $[\texttt{BrC}]^{\downarrow}$ message. Each message contains at most a VRM, which is $O(d^2)$. □

Lemma 1. *The DCOP is arc consistent after the AC propagation phase.*

Proof. We prove this result by contradiction. Assume that there are $a_i, a_j \in \mathbf{A}$ and $a \in D_i$ such that $\forall b \in D_j$, $(a, b) \notin f_{ij}$. Let $b_1, \ldots, b_m$ be all the (pruned) values in D_j supporting a. We have the following two cases:

- $a_i \in P_{a_j} \cup PP_{a_j}$. If agent a_j pruned all its values b_r $(1 \leq r \leq m)$ from D_j, then the value a is pruned from the copy of the domain D_i held at a_j ($\hat{\mathbf{M}}_{ji|i}$ will not include the value a) (line 22). When a_i receives an $AC^{\uparrow}$ message from each $a_k \in C_{a_i} \cup PC_{a_i}$ (including a_j), it updates its own domain with the copy received from each agent (lines 20-21) removing a from D_i and resulting in a contradiction.
- $a_i \in C_{a_j} \cup PC_{a_j}$. Agent a_j can prune all its values b_r $(1 \leq r \leq m)$ from D_j in the following two ways. In case 1, agent a_i prunes all the values b_r from a copy of D_j during its AC consistency enforcement (line 22), sends up an $AC^{\uparrow}$ message to a_j, and a_j prunes all its values b_r from its D_j. However, in this case, agent a_i would have also pruned value a from its domain, resulting in a contradiction. In case 2, some other agent a_k that shares a constraint f_{kj} with agent a_j prunes all the values b_r from the copy of D_j during its AC consistency enforcement, sends up an $AC^{\uparrow}$ message to a_j, and a_j prunes all its values b_r from its D_j. In this case, a_j will eventually send an $AC^{\downarrow}$ message to a_i that contains its updated domain without the values b_r. Then, agent a_i will prune value a from its domain in its AC consistency enforcement (line 22), resulting in a contradiction.

□

Lemma 2. *The DCOP is branch consistent after the BrC propagation phase.*

Proof. We prove by induction on the number of variables in the paths $x_i = x_{k_1}, \ldots, x_{k_m} = x_j$, such that $x_{k_1} \succ \ldots \succ x_{k_m}$.
Base Case $(m = 2)$**:** We know that $x_j \in C_{a_i}$ and there is only one path from x_i to x_j via the constraint f_{ij}. Additionally, this constraint is arc consistent because the BrC propagation phase runs after the AC propagation phase. Thus, all the remaining pairs of values in both variables are by definition branch consistent (Definition 10). The VRM M_{ji} is thus branch consistent.
Induction Assumption: Assume that for any $2 \leq q \leq r$ and paths $x_i = x_{k_1}, \ldots, x_{k_q} = x_j$ with $x_{k_1} \succ \ldots \succ x_{k_q}$, there is a VRM M_{ji} that is branch consistent.
Induction Case $(m = r+1)$**:** We know that the paths from $x_i = x_{k_1}$ to x_{k_r} is branch consistent from the induction assumption. Thus, the VRM $M_{k_r k_1}$ received by $x_{k_{r+1}}$ is branch consistent. Additionally, all the constraints between any x_{k_p} $(1 \leq p \leq r)$ and $x_{k_{r+1}}$ are arc consistent because the BrC propagation phase runs after the AC propagation phase. Thus, the VRMs $\hat{M}_{k_{r+1} k_p}$ are also branch consistent.
We now show that the algorithm removes values of $x_{k_{r+1}}$ that are not branch consistent with values of its ancestors in the following two cases:

- For paths that include the constraint between x_r and x_{r+1}, BrC-DPOP takes the regular product (line 42), which removes all inconsistent values.
- For paths that do not include the constraint between x_r and x_{r+1} and, thus, must include the constraint between x_{k_1} and $x_{k_{r+1}}$, BrC-DPOP performs the entrywise product (line 43), which removes all inconsistent values.

□

Theorem 7. *BrC-DPOP is complete and correct.*

Proof. The completeness and correctness of BrC-DPOP follows from the correctness and completeness of DPOP [96] and the correctness and completeness of the AC and BrC propagation phases (Theorems 5, 6, and Lemmas 1, and 2). □

Corollary 2. *Both the UTIL and the VALUE phases require $O(n)$ number of messages.*

Corollary 3. *The memory requirement of BrC-DPOP is in the worst case exponential in the induced width of the problem for each agent.*

Both corollaries follow trivially from the network load and memory requirements of DPOP, since no values are pruned from the AC and BrC propagation phases in the worst case.

4.2.4 Related Work

We characterize the approaches that prune values of variables in DCOPs along two general types. Algorithms in the first category *propagates exclusively hard constraints* (BrC-DPOP falls into this category). To the best of our knowledge, the only existing work that falls into this category is H-DPOP [63], which, like BrC-DPOP, is also an extension of DPOP. The main difference between H-DPOP and BrC-DPOP is that instead of VRMs, each agent a_i in H-DPOP uses *constraint decision diagrams* (CDDs) to represent the space of possible value assignments of variables in its separator set $sep(a_i)$. A CDD is a rooted directed acyclic graph structured by levels, one for each variable in $sep(a_i)$. In each level, a non-terminal node represents a possible value assignment for the associated variable. Each non-terminal node v has a list of successors: one for each value u in the next variable for which the pair (u, v) is satisfied by the constraint between the two variables. As a result of using CDDs, H-DPOP suffers from two limitations:

1. H-DPOP can be slower than DPOP because maintaining and performing join and projection operations on CDD are computationally expensive. In contrast, maintaining and performing operations on VRMs can be faster, which we will demonstrate in the experimental results section later.
2. H-DPOP cannot fully exploit information of hard constraints to reduce the size of UTIL messages.

Consider the DCOP instance of Figure 4.2, where the domains for the variables x_1, x_3, x_4, and x_5 are represented by the set $\{1, \dots, 100\}$, while the domain for variable x_2 is the set $\{1, 2\}$. In H-DPOP, a_5 is not aware of the constraints $x_1 < x_2$ and $x_1 < x_3$—neither x_2 nor x_3 are in $sep(a_5)$, thus no pruning will be enforced. Its UTIL table will hence contain $100^2 = 10,000$ utilities for each combination of values of x_4 and x_1. This is the same table that DPOP would construct. In contrast, in BrC-DPOP, the domains of x_1 and x_2 will be pruned to $\{1\}$ and $\{2\}$, respectively, and the domains of x_3, x_4, and x_5 to $\{2, \dots, 100\}$. Therefore, the UTIL table that a_5 sends to a_4 contains $99 \times 1 = 99$ utilities. Aside from these two limitations, a more critical limitation of H-DPOP is its assumption that each agent has knowledge of all the constraints whose scope is a subset of its separator set. This assumption is stronger than the assumptions made by most DCOP algorithms and might cause privacy concerns in some applications. In contrast, BrC-DPOP does not make such assumptions.

Algorithms in the second category *propagates lower and upper bounds*. Researchers have extended search-based DCOP algorithms (e.g., BnB-ADOPT and its enhanced versions [121, 48, 51]) to maintain soft-arc consistency in a distributed manner [9, 49, 47]. Such techniques are typically very effective in search-based algorithms as their runtime depends on the accuracy of its lower and upper bounds.

Finally, it is important to note the differences between branch consistency and path consistency [82]. One can view branch consistency as a weaker version of path consistency, where all the variables in a path must be ordered according to the relation $\prec$, and only a subset of all possible paths have to be examined for consistency. Thus, one can view branch consistency as a form of consistency tailored to pseudo-trees, where each agent can only communicate with neighboring agents. [19] One of the straightforward ways to enforce path consistency in a DCOP would require that all agents are able to communicate with every other agent. Unfortunately, this requirement would violate the common assumption that agents can only communicate with neighboring agents in a DCOP.

4.2.5 Experimental Evaluation

We implemented a variant of BrC-DPOP, called AC-DPOP, that enforces arc consistency only in order to assess the impact of the branch consistency phase in BrC-DPOP. Moreover, in order to be as comprehensive as possible in our evaluations, we also implemented a variant of H-DPOP called PH-DPOP, which stands for Privacy-based H-DPOP, that restricts the amount of information that each agent can access to the amount common in most DCOP algorithms including BrC-DPOP. Specifically, agents in PH-DPOP can only access their own constraints and, unlike H-DPOP, cannot access their neighboring agents' constraints.

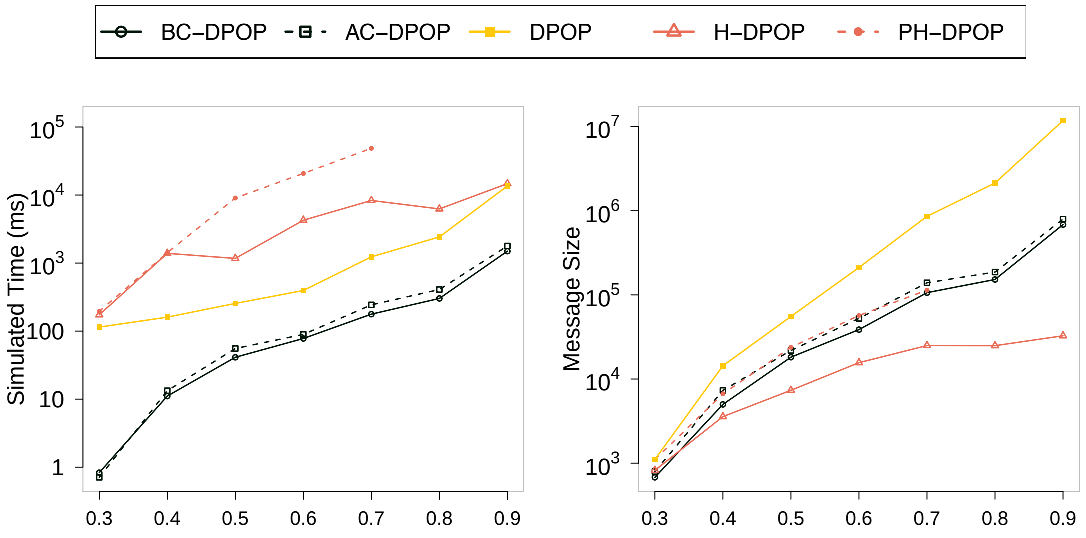
In our experiments,[3] we compare AC-DPOP and BrC-DPOP against DPOP [96], H-DPOP [63], and PH-DPOP. We use a publicly-available implementation of DPOP available in the FRODO framework [68] and an implementation of H-DPOP provided by the authors. We ensure that all algorithms use the same pseudo-tree for fair comparisons. All experiments are performed on an Intel i7 Quadcore 3.4GHz machine with 16GB of RAM with a 300-second timeout. If an algorithm fails to solve a problem, it is due to either memory limitations or timeout. We conduct our experiments on random graphs [30], where we systematically vary the constraint density p_1 and constraint tightness p_2,[4] and distributed Radio Link Frequency Assignment (RLFA) problems [15], where we vary the number of agents $|\mathbf{A}|$ in the problem. We generated 50 instances for each experimental setting, and we report the average runtime, measured using the simulated runtime metric [111], and the average total message size, measured in the number of utility values in the UTIL tables. For the distributed RLFA problems, we also report the percentage of satisfiable instances solved to show the scalability of the algorithms.

Random Graphs: In our experiments, we set $|\mathbf{A}| = 10$, $|\mathbf{X}| = 10$, $|D_i| = 8$ for all variables. We vary p_1 (while setting $p_2 = 0.6$) and vary the p_2 (while setting $p_1 = 0.6$). We did not bound the tree-width, which is determined based on the underlying graph and randomly generated. We used hard constraints that are either the "less than" or "different" constraints. We also assign a unary constraint to each variable that gives it a utility corresponding to each its value assignments.

Figures 4.3, 4.4, and 4.5, show the runtimes and the message size of the algorithms at varying of, repsectively, the constraint graph density p_1, the constraint graph tightness p_2, and the number of agents $\mathbf{A}$. We omit results of an algorithm for a specific parameter if it fails to solve 50% of the satisfiable instances for that parameter. We make the following observations:

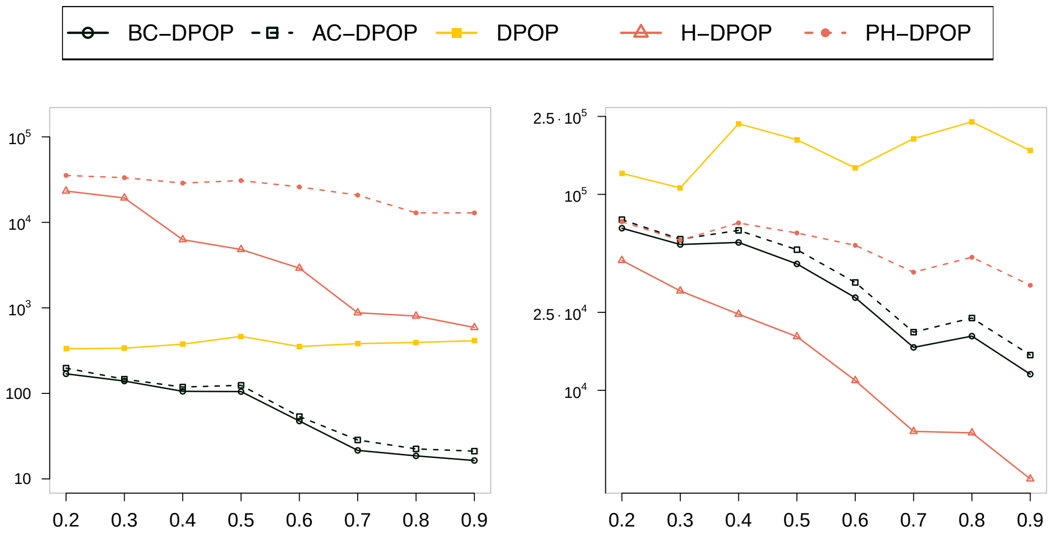
[3] available at http://www.cs.nmsu.edu/klap/brc-dpop_cp14/

[4] p_1 and p_2 are defined as the ratio between the number of binary constraints in the problem and the maximum possible number of binary constraints in the problem and the ratio between the number of hard constraints and the number of constraints in the problem, respectively.



(a) Random Graphs: Varying p1 (d) Random Graphs: Varying p1

Figure 4.3: Runtimes and Message Sizes at varying of the constraint graph density p_1.



(b) Random Graphs: Varying p2 (e) Random Graphs: Varying p2

Figure 4.4: Runtimes and Message Sizes at varying of the constraint tightness p_2.

- On message sizes, BrC-DPOP uses smaller messages than AC-DPOP because BrC-DPOP prunes more values due to its BrC propagation enforcement. H-DPOP uses smaller messages than BrC-DPOP and AC-DPOP because agents in H-DPOP utilize more information about the neighbors' constraints to prune values. In contrast, agents in BrC-DPOP and AC-DPOP only utilize information on their own constraints to prune values. BrC-DPOP and AC-DPOP use smaller messages than PH-

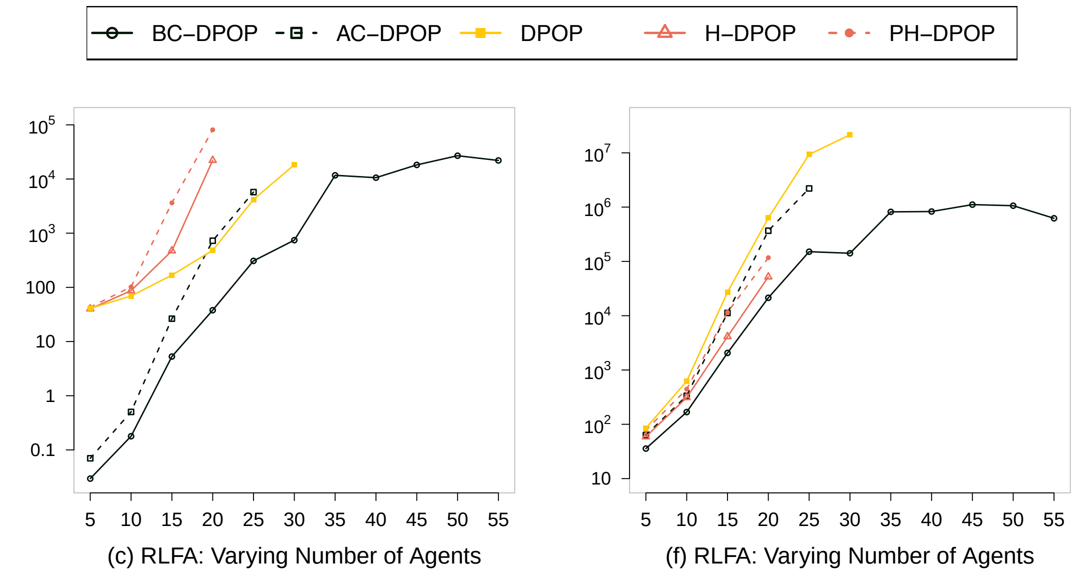


Figure 4.5: Runtimes and Message Sizes at varying of the number of agents $\mathbf{A}$.

DPOP at large p_2 values, highlighting the strength of the AC and BrC propagation phases compared to the pruning techniques in PH-DPOP. Finally, they all use smaller messages than DPOP because they all prune values while DPOP does not.

- On runtimes, BrC-DPOP is slightly faster than AC-DPOP because BrC-DPOP prunes more values than AC-DPOP. Additionally, these results also indicate that the overhead of the BrC propagation phase is relatively small. BrC-DPOP and AC-DPOP are faster than DPOP because they do not need to perform operations on the pruned values. This also indicates that the overhead of the AC propagation phase is small. In our experiments, the number of [AC] messages exchanged during the AC propagation phase never exceeds $3|\mathbf{F}|$ and is, on average, as small as $|\mathbf{F}|$. DPOP is faster than H-DPOP and PH-DPOP because they maintain and perform operations on CDDs, which are computationally very expensive. In contrast, BrC-DPOP maintains and performs operations on matrices, which are more computationally efficient.

Distributed RLFA Problem: In these problems, we are given a set of links $\{L_1, \ldots, L_r\}$, each consisting of a transmitter and a receiver. Each link must be assigned a frequency from a given set F. At the same time the total interference at the receivers must be reduced below an acceptable level using as few frequencies as possible. In our model, each transmitter corresponds to an agent (and a variable). The domain of each variable consists of the frequencies that can be assigned to the corresponding transmitter. The interference between transmitters are modeled as constraints of the form $|x_i - x_j| > s$, where x_i and x_j are variables and $s \geq 0$ is a randomly generated required frequency separation. We also assign a utility value to each frequency taken by each agent, represented as a unary soft constraint, which represents a preference for an agent to transmit at a given frequency.

For generating the constraint graphs, we vary $|\mathbf{A}|$ and fix the other parameters: $|D_i| = 6, p_2 = 0.55, s \in \{2, 3\}$. We also set the maximum number of neighbors for each agent to 3 in order to generate

$\|\mathbf{A}\|$	5	10	15	20	25	30	35	40	45	50	55
BrC-DPOP	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.97**	**0.52**	**0.78**	**0.73**	**0.70**	**0.51**
AC-DPOP	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.39	0.11	0.30	0.15	0.15	0.19
H-DPOP	**1.00**	**1.00**	**1.00**	**1.00**	0.46	0.12	0.00	0.00	0.00	0.00	0.00
PH-DPOP	**1.00**	**1.00**	**1.00**	**1.00**	0.21	0.09	0.00	0.00	0.00	0.00	0.00
DPOP	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.67	0.23	0.35	0.23	0.29	0.19

Table 4.1: Percentage of Satisfiable Instances Solved

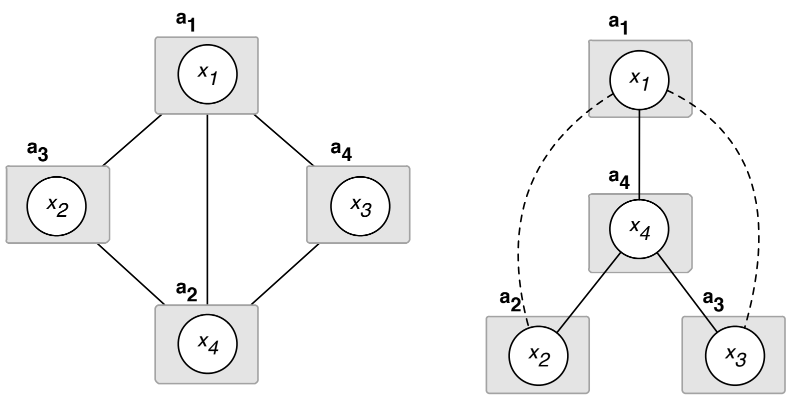
more satisfiable instances. Figure 4.5(c) shows the runtimes and Figure 4.5(f) shows the message sizes. We omit results of an algorithm for a specific parameter if it fails to solve 50% of the satisfiable instances for that parameter.

We observe trends that are similar to those in the earlier random graphs except that the message size of H-DPOP is slightly larger than of those of BrC-DPOP. Therefore, as we have described in section 4.2.4, it is possible for H-DPOP to prune fewer values despite using more information. Additionally, both H-DPOP and PH-DPOP can only solve small problems and failed to solve some problems that DPOP successfully solved. Table 4.1 tabulates the percentage of satisfiable problem instances solved by each algorithm (the largest percentage in each parameter setting is shown in bold), where it is clear that BrC-DPOP is more scalable than all its counterparts.

4.3 Distributed Large Neighborhood Search

In this section we describe the *Distributed Large Neighborhood Search (D-LNS)* framework, which aims at providing an incomplete solution to solve DCOPs. D-LNS solves DCOPs by building on the strengths of *centralized* LNS [109], a *centralized* meta-heuristic that iteratively explores complex neighborhoods of the search space to find better candidate solutions. LNS has been shown to be very effective in solving a number of optimization problems [41, 106]. While typical LNS approaches focus on iteratively refining lower bounds of a solution, we propose a method that can iteratively refine both lower and upper bounds of a solution, imposing no restrictions (i.e., linearity or convexity) on the objective function and constraints. To illustrate the generality of the proposed framework we describe two distributed search algorithms, DBR-DPOP and T-DBR, built within the D-LNS framework, and characterized by the ability to exploit problem structure and offer low network usage—T-DBR provides also a low computational complexity per agent. Our evaluation against representatives of search-based, inference-based, and region-optimal-based incomplete DCOP algorithms shows that T-DBR converges faster to better solutions, provides tighter solution quality bounds, and is more scalable.

Through the section, we will use the example DCOP shown in Figure 4.6 to illustrate the execution of T-DBR on D-LNS, refining both upper and lower bounds of the solution found during the iterative process. Fig. 4.6(a) shows the graph of a DCOP with agents $a_1, \dots, a_4$, each controlling a variable with domain $\{0,1\}$. Fig. 4.6(b) shows a pseudo-tree (solid lines are tree edges, dotted lines are backedges). Fig. 4.6(c) shows the utilities.



(a) Constraint Graph (b) Pseudo-tree

$j = 2,3$

x_1	x_j	util
0	0	10
0	1	0
1	0	2
1	1	0

$j = 1,2,3$

x_j	x_4	util
0	0	0
0	1	6
1	0	0
1	1	10

(c) Constraints

Figure 4.6: Example DCOP

4.3.1 Notation and Definitions

Given a DCOP P, in this section, we denote with $G^k = \langle X^k, E^k \rangle$, the subgraph of G_P used in the execution of our iterative algorithms, where $X^k \subseteq \mathbf{X}$ and $E^k \subseteq E_C$.

Large Neighborhood Search. In (*centralized*) *Large Neighborhood Search (LNS),* an initial solution is iteratively improved by repeatedly *destroying* it and *repairing* it. Destroying a solution means selecting a subset of variables whose current values will be discarded. The set of such variables is referred to as *large neighborhood (LN).* Repairing a solution means finding a new value assignment for the destroyed variables, given that the other non-destroyed variables maintain their values from the previous iteration.

The peculiarity of LNS, compared to other local search techniques, is the (larger) size of the neighborhood to explore at each step. It relies on the intuition that searching over a larger neighborhood allows the process to escape local minima and find better candidate solutions.

4.3.2 DLNS Framework and Repair Algorithms

In this section, we introduce D-LNS, a general distributed LNS framework to solve DCOPs. Our D-LNS solutions need to take into account factors that are critical for the performance of distributed systems, such as network load (i.e., number and size of messages exchanged by agents) and the restriction that each agent is only aware of its local subproblem (i.e., its neighbors and the constraints whose scope includes its variables). Such properties make typical centralized LNS techniques unsuitable and infeasible for DCOPs.

Algorithm 3 shows the general structure of D-LNS, as executed by each agent $a_i \in \mathbf{A}$. After initializing its iteration counter k (line 1), its current value assignment $\mathbf{x}_i^0$ (done by randomly assigning values to variables or by exploiting domain knowledge when available), and its current lower and upper bounds LB_i^0 and UB_i^0 of the optimal utility (line 2), the agent, like in LNS, iterates through the

Algorithm 3: D-LNS

```
1 $k \leftarrow 0$;
2 $\langle \mathbf{x}_i^0, LB_i^0, UB_i^0 \rangle \leftarrow$ VALUE-INITIALIZATION();
3 while termination condition is not met do
4     $k \leftarrow k + 1$;
5     $z_i^k \leftarrow$ DESTROY-ALGORITHM();
6     if $z_i^k = \circ$ then $\mathbf{x}_i^k \leftarrow NULL$;
7         else $\mathbf{x}_i^k \leftarrow \mathbf{x}_i^{k-1}$ $\langle \mathbf{x}_i^k, LB_i^k, UB_i^k \rangle \leftarrow$ REPAIR-ALGORITHM($z_i^k$);
8     if not Accept ($\mathbf{x}_i^k, \mathbf{x}_i^{k-1}$) then $\mathbf{x}_i^k \leftarrow \mathbf{x}_i^{k-1}$
```

destroy and repair phases (lines 3-8) until a termination condition occurs (line 3). Possible termination conditions include reaching a maximum value of k, a timeout limit, or a confidence threshold on the error of the reported best solution.

Destroy Phase. The result of this phase is the generation of a LN, which we refer to as $LN^k \subseteq \mathbf{X}$, for each iteration k. This step is executed in a distributed fashion, having each agent a_i calling a DESTROY-ALGORITHM to determine if its local variable x_i should be ***destroyed*** ($\circ$) or ***preserved*** ($\star$), as indicated by the flag z_i^k (line 5). We say that destroyed (resp. preserved) variables are (resp. are not) in LN^k. In a typical destroy process, such decisions can be either random or made by exploiting domain knowledge. For example, in a scheduling problem, one may choose to preserve the start times of each activity and destroy the other variables. D-LNS allows the agents to use any destroy schema to achieve the desired outcome. Once the destroyed variables are determined, the agents reset their values and keep the values of the preserved variables from the previous iteration (line 6).

Repair Phase. The agents start the repair phase, which seeks to find new value assignments for the destroyed variables, by calling a REPAIR-ALGORITHM (line 7). The goal of this phase is to find an improved solution by searching over a LN. D-LNS is general in that it does not impose any restriction on the way agents coordinate to solve this problem. We propose two distributed repair algorithms in the next section, that provide quality guarantees and online bound refinements. Once the agents find and evaluate a new solution, they either accept it or reject it (line 8). While most of the current incomplete DCOP algorithms fail to guarantee the consistency of the solution returned w.r.t. the hard constraints of the problem [91], D-LNS can accommodate consistency checks during the repair phase.

Distributed Bounded Repair

We now introduce the *Distributed Bounded Repair* (DBR), a general REPAIR algorithm framework that iteratively refines the lower and upper bounds of the DCOP solution. Its general structure is illustrated in the flow chart of Figure 4.7. At each iteration k, each *DBR* agent checks if its local variable was preserved or destroyed. In the former case, the agent waits for the conditions to start the *Bounding* phase, which is algorithm dependent. In the latter case the agent executes, in order, the following phases:

Relaxation Phase. Given a DCOP P, this phase constructs two *relaxations* of P, $\check{P}^k$ and $\hat{P}^k$, which are used to compute, respectively, a lower and an upper bound on the optimal utility for P. Let $G^k = \langle LN^k, E^k \rangle$ be the subgraph of G_P in iteration k, where $E^k = \{(x,y) \mid (x,y) \in E_C; x,y \in LN^k\}$ is the subset of edges of E_C whose elements involve exclusively nodes in LN^k. Both problem relaxations $\check{P}^k$ and $\hat{P}^k$ are solved using the same underlying ***relaxation graph*** $\tilde{G}^k = \langle LN^k, \tilde{E}^k \rangle$, computed from G^k, where $\tilde{E}^k \subseteq E^k$ depends on the algorithm adopted.

In the problem $\check{P}^k$, we wish to find a solution $\check{\mathbf{x}}^k$ using

$$\begin{aligned}\check{\mathbf{x}}^k &= \operatorname*{argmax}_{\mathbf{x}} \check{F}^k(\mathbf{x}) \qquad (4.1)\\ &= \operatorname*{argmax}_{\mathbf{x}} \sum_{f \in \tilde{E}^k} f(\mathbf{x}_i, \mathbf{x}_j) \;+\; \sum_{\substack{f \in \mathcal{F},\, \mathbf{x}^f = \{x_i, x_j\} \\ x_i \in LN^k,\, x_j \notin LN^k}} f(\mathbf{x}_i, \check{\mathbf{x}}_j^{k-1})\end{aligned}$$

where $\check{\mathbf{x}}_j^{k-1}$ is the value assigned to the preserved variable x_j for problem $\check{P}^{k-1}$ in the previous iteration. The first summation is over all functions listed in $\tilde{E}^k$, while the second is over all functions between destroyed and preserved variables. Thus, solving $\check{P}^k$ means optimizing over all the destroyed variables given that the preserved ones take on their previous value, and ignoring the (possibly empty) set of edges $E_C \setminus \tilde{E}^k \cup \{(x, y) \mid (x, y) \in E_C; x \in LN^k, y \notin LN^k\}$ that are not part of the relaxed graph. This solution is used to compute lower bounds during the *bounding phase*.

In the problem $\hat{P}^k$, we wish to find a solution $\hat{\mathbf{x}}^k$ using

$$\hat{\mathbf{x}}^k = \operatorname*{argmax}_{\mathbf{x}} \hat{F}^k(\mathbf{x}) = \operatorname*{argmax}_{\mathbf{x}} \sum_{f \in \mathbf{F}} \hat{f}^k(\mathbf{x}_i, \mathbf{x}_j) \qquad (4.2)$$

where $\hat{f}^k(\mathbf{x}_i, \mathbf{x}_j)$ is defined as:

$$\hat{f}^k(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \max\limits_{d_i \in D_i, d_j \in D_j} f(d_i, d_j) & \text{if } \Gamma_f^k = \emptyset \\ \max\left\{ \frac{\tilde{F}^k}{|\tilde{E}^k|}, \max\limits_{\ell \in \Gamma_f^{k-1}} \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell) \right\} & \text{if } f \in \tilde{E}^k \\ \hat{f}^{k-1}(\hat{\mathbf{x}}_i^{k-1}, \hat{\mathbf{x}}_j^{k-1}) & \text{otherwise} \end{cases}$$

where $\tilde{F}^k = \max_{\mathbf{x}} \sum_{f \in \tilde{E}^k} f(\mathbf{x}_i, \mathbf{x}_j)$, and Γ_f^k is the set of past iteration indices for which the function f was an edge in the relaxation graph. Specifically,

$$\Gamma_f^k = \left\{ \ell \mid f \in \tilde{E}^\ell \wedge 0 < \ell \leq k \right\} \qquad (4.3)$$

Therefore, the utility of $\hat{F}^k(\hat{\mathbf{x}}^k)$ is composed of three parts.

- The first part involves all functions that have never been part of $\tilde{E}^k$ up to the current iteration,
- The second part involves all the functions optimized in the current iteration, and
- The third part involves all the remaining functions.

The utility of each function in the first part is the maximal utility over all possible pairs of value combinations of variables in the scope of that function. The utility of each function in the second part is the largest utility among the mean utility of the functions optimized in the current iteration (i.e., those in $\tilde{E}^k$), and the utilities of such function optimized in a past iteration. The utility of each function in the third part is equal to the utility assigned to such function in the previous iteration. In particular, imposing that the edges optimized in the current iteration contribute at most equally (i.e., as the mean utility of $\tilde{F}^k$) to

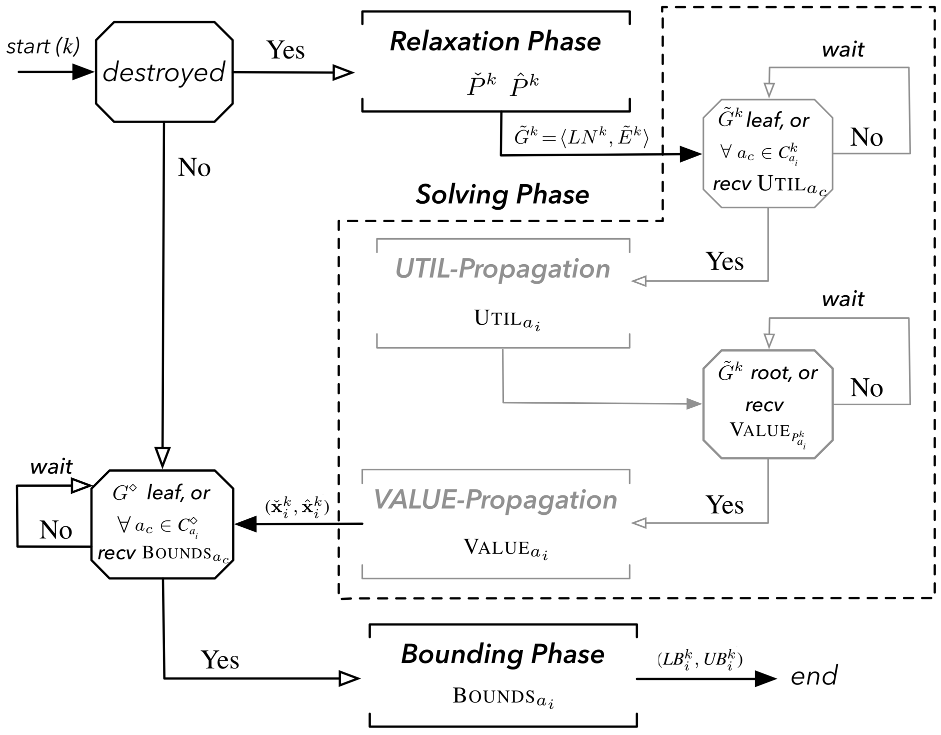

start (k)
destroyed
Yes
No
Relaxation Phase
$\check{P}^k$ $\hat{P}^k$
$\tilde{G}^k = \langle LN^k, \tilde{E}^k \rangle$
wait
$\tilde{G}^k$ leaf, or $\forall a_c \in C^k_{a_i}$ recv UTIL_{a_c}
No
Solving Phase
UTIL-Propagation
UTIL_{a_i}
Yes
wait
$\tilde{G}^k$ root, or recv $\text{VALUE}_{P^k_{a_i}}$
No
VALUE-Propagation
VALUE_{a_i}
Yes
wait
No
$G^\diamond$ leaf, or $\forall a_c \in C^\diamond_{a_i}$ recv BOUNDS_{a_c}
$(\check{x}^k_i, \hat{x}^k_i)$
Yes
Bounding Phase
BOUNDS_{a_i}
(LB^k_i, UB^k_i)
end

Figure 4.7: DBR Flow chart. The *Solving phase* illustrates the T-DBR algorithm's solving phase.

the final utility of $\hat{P}^k$ allows us to not underestimate the solution upper bound within the iterative process (see Lemma 3). In summary, solving $\hat{P}^k$ means finding the solution $\hat{\mathbf{x}}^k$ that maximizes $\hat{F}^k(\hat{\mathbf{x}}^k)$. This solution is used to compute upper bounds during the *bounding phase*.

Solving Phase. Next, DBR solves the relaxed DCOPs $\check{P}^k$ and $\hat{P}^k$ using the equations above. At a high-level, one can use any complete DCOP algorithm to solve $\check{P}^k$ and $\hat{P}^k$. Below, we describe two inference-based DBR algorithms, defined over different relaxation graphs $\tilde{G}^k$. Thus, the output of this phase are the values for the agent's local variable, $\check{\mathbf{x}}^k_i, \hat{\mathbf{x}}^k_i$, associated to eq. (1) and (2).

Bounding Phase. Once the relaxed problems are solved, all DBR agents are ready to start the Bounding phase. Such phase results in computing the lower and upper bounds based on the solutions $\check{\mathbf{x}}^k$ and $\hat{\mathbf{x}}^k$. As we show in Theorems 8 and 9, $\mathbf{F}_g(\check{\mathbf{x}}^k) \leq \mathbf{F}_g(\mathbf{x}^*) \leq \hat{F}^k(\hat{\mathbf{x}}_k)$. Therefore, $\rho = \frac{\min_k \hat{F}^k(\hat{\mathbf{x}}^k)}{\max_k \mathbf{F}_g(\check{\mathbf{x}}^k)}$ is a guaranteed approximation ratio for P with solution $\mathbf{x} = \arg\max_{\check{\mathbf{x}}_k} \mathbf{F}_g(\check{\mathbf{x}}^k)$.

The significance of this Repair framework is that it enables our D-LNS to iteratively refine both lower and upper bounds of the solution, without imposing any restrictions on the form of the objective function and of the constraints adopted. Below, we introduce two implementations of the DBR framework, both summarized in the flow-chart of Figure 4.7, whose solving phase is shown in the dotted area.

DBR-DPOP Algorithm

DBR-DPOP solves the relaxed DCOPs $\check{P}^k$ and $\hat{P}^k$ over the relaxed graph $\tilde{G}^k = \langle LN^k, E^k \rangle$. Thus, $\tilde{E}^k = E^k$, and solving problem $\check{P}^k$ means optimizing over all the destroyed variables ignoring no edges in E^k.

The DBR-DPOP *solving phase* uses DPOP [96], a complete inference-based algorithm composed of two phases operating on a DFS pseudo-tree.

- In the ***utility propagation phase***, each agent, starting from the leaves of the pseudo-tree, projects out its own variable and sends its projected utilities to its parent. These utilities are propagated up the pseudo-tree induced from $\tilde{G}^k$ until they reach the root. The hard constraints of the problem can be naturally handled in this phase, by pruning all inconsistent values before sending a message to its parent.
- Once the root receives utilities from all its children, it starts the ***value propagation phase***, where it selects the value that maximizes its utility and sends it to its children, which repeat the same process. The problem is solved as soon as the values reach the leaves.

Note that the relaxation process may create a forest, in which case one should execute the algorithm in each tree of the forest. As a technical note, DBR-DPOP solves the two relaxed DCOPs in parallel. In the utility propagation, each agent computes two sets of utilities, one for each relaxed problem, and sends them to its parent. In the value propagation phase, each agent selects two values, one for each relaxed problem, and sends them to its children.

DBR-DPOP has the same worst case order complexity of DPOP, that is, exponential in the induced width of the relaxed graph $\tilde{G}^k$. Thus, we introduce another algorithm characterized by a smaller complexity and low network load.

Tree-based DBR Algorithm

Tree-based DBR (T-DBR) defines the relaxed DCOPs $\check{P}^k$ and $\hat{P}^k$ using a pseudo-tree structure $T^k = \langle LN^k, E_{T^k} \rangle$ that is computed from the subgraph G^k. Thus, $\tilde{E}^k = E_{T^k}$, and solving problem $\check{P}^k$ means optimizing over all the destroyed variables ignoring backedges. Its general solving schema is similar to that of DPOP, in that it uses Utility and Value propagation phases; however, the different underlying relaxation graph adopted imposes several important differences. Algorithm 2 shows the T-DBR pseudocode. We use the following notations: $P^k_{a_i}$, $C^k_{a_i}$, $PP^k_{a_i}$ denote the parent, the set of children, and pseudo-parents of the agent a_i, at iteration k. The set of these items is referred to as $\mathbf{T}^k_{a_i}$, which is a_i's *local view* of the pseudo-tree T^k. We use "$^\diamond$" to refer to the items associated with the pseudo-tree $T^\diamond$. $\check{\chi}_{a_i}$ and $\hat{\chi}_{a_i}$ denote a_i's *context* (i.e., the values for each $x_j \in N_{a_i}$) with respect to problems $\check{P}$ and $\hat{P}$, respectively. We assume that by the end of the destroy phase (line 6) each agent knows its current context as well as which of its neighboring agents has been destroyed or preserved. In each iteration k, T-DBR executes the following phases:

Relaxation Phase. It constructs a pseudo-tree T^k (line 9), which ignores, from $G^\diamond$, the destroyed variables as well as the functions involving these variables in their scopes. The construction prioritizes tree-edges that have not been chosen in previous pseudo-trees over the others.

Solving Phase. Similarly to DPOP-DBR, T-DBR solving phase is composed of two phases operating on

```
Algorithm 2: T-DBR($z_i^k$)
9  $\mathbf{T}_{a_i}^k \leftarrow$ RELAXATION($z_i^k$)
10 UTIL-PROPAGATION($\mathbf{T}_{a_i}^k$)
11 $\langle \check{\chi}_{a_i}^k, \hat{\chi}_{a_i}^k \rangle \leftarrow$ VALUE-PROPAGATION($\mathbf{T}_{a_i}^k$)
12 $\langle LB_i^k, UB_i^k \rangle \leftarrow$ BOUND-PROPAGATION($\check{\chi}_{a_i}^k, \hat{\chi}_{a_i}^k$)
13 return $\langle \check{\mathbf{x}}_i^k, LB_i^k, UB_i^k \rangle$
```

```
Procedure UTIL-Propagation($\mathbf{T}_{a_i}^k$)
14 receive UTIL$_{a_c}$($\check{U}_{a_c}$, $\hat{U}_{a_c}$) from each $a_c \in C_{a_i}^k$
15 forall the values $\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}$ do
16     $\check{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}) \leftarrow f(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}) + \sum_{a_c \in C_{a_i}^k} \check{U}_{a_c}(\mathbf{x}_i) + \sum_{x_j \notin LN^k} f(\mathbf{x}_i, \check{\mathbf{x}}_j^{k-1})$
17     $\hat{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}) \leftarrow f(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}) + \sum_{a_c \in C_{a_i}^k} \hat{U}_{a_c}(\mathbf{x}_i)$
18 forall the values $\mathbf{x}_{P_{a_i}^k}$ do
19     $\langle \check{U}'_{a_i}(\mathbf{x}_{P_{a_i}^k}), \hat{U}'_{a_i}(\mathbf{x}_{P_{a_i}^k}) \rangle \leftarrow \langle \max_{\mathbf{x}_i} \check{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}), \max_{\mathbf{x}_i} \hat{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k}) \rangle$
20 send UTIL$_{a_i}$($\check{U}'_{a_i}$, $\hat{U}'_{a_i}$) msg to $P_{a_i}^k$
```

```
Function VALUE-Propagation($\mathbf{T}_i^k$)
21 if $P_{a_i}^k = NULL$ then
22     $\langle \check{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k \rangle \leftarrow \langle \operatorname{argmax}_{\mathbf{x}_i} \check{U}_{a_i}(\mathbf{x}_i), \operatorname{argmax}_{\mathbf{x}_i} \hat{U}_{a_i}(\mathbf{x}_i) \rangle$
23     send VALUE$_{a_i}$($\check{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k$) msg to each $a_j \in N_{a_i}$
24     forall the $a_j \in N_{a_i}$ do
25         receive VALUE$_{a_j}$($\check{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k$) msg from $a_j$
26         Update $x_j$ in $\langle \check{\chi}_{a_i}^k, \hat{\chi}_{a_i}^k \rangle$ with $\langle \check{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k \rangle$
27 else
28     forall the $a_j \in N_{a_i}$ do
29         receive VALUE$_{a_j}$($\check{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k$) msg from $a_j$
30         Update $x_j$ in $\langle \check{\chi}_{a_i}^k, \hat{\chi}_{a_i}^k \rangle$ with $\langle \check{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k \rangle$
31         if $a_j = P_{a_i}^k$ then
32             $\langle \check{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k \rangle \leftarrow \langle \operatorname{argmax}_{\mathbf{x}_i} \check{U}_{a_i}(\mathbf{x}_i), \operatorname{argmax}_{\mathbf{x}_i} \hat{U}_{a_i}(\mathbf{x}_i) \rangle$
33             send VALUE$_{a_i}$($\check{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k$) msg to each $a_j \in N_{a_i}$
34 return $\langle \check{\chi}_{a_i}^k, \hat{\chi}_{a_i}^k \rangle$
```

the relaxed pseudo-tree T^k, and executed synchronously:

- *Utility Propagation Phase.* After the pseudo-tree T^k is constructed (line 10), each leaf agent com-

```
Procedure BOUND-Propagation($\check{\chi}_i^k, \hat{\chi}_i^k$)
35 receive BOUNDS$_{a_c}$($LB_c^k, UB_c^k$) msg from each $a_c \in C_{a_i}^{\diamond}$
36 $LB_i^k \leftarrow f(\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_{P_{a_i}^{\diamond}}^k) + \sum_{a_j \in PP_{a_i}^{\diamond}} f(\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_j^k) + \sum_{a_c \in C_{a_i}^{\diamond}} LB_c^k$
37 $UB_i^k \leftarrow \hat{f}^k(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{P_{a_i}^{\diamond}}) + \sum_{a_j \in PP_{a_i}^{\diamond}} \hat{f}^k(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j) + \sum_{a_c \in C_{a_i}^{\diamond}} UB_c^k$
38 send BOUNDS$_{a_i}$($LB_i^k, UB_i^k$) msg to $P_{a_i}^{\diamond}$
```

putes the optimal sum of utilities in its subtree considering exclusively tree edges (i.e., edges in E_{T^k}) and edges with destroyed variables. Each leaf agent computes the utilities $\check{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k})$ and $\hat{U}_{a_i}(\mathbf{x}_i, \mathbf{x}_{P_{a_i}^k})$ for each pair of values of its variable $\mathbf{x}_i$ and its parent's variable $\mathbf{x}_{P_{a_i}^k}$ (lines 15-17), in preparation for retrieving the solutions of $\check{P}$ and $\hat{P}$, used during the bounding phase. The agent projects itself out (lines 18-19) and sends the projected utilities to its parent in a UTIL message (line 20). Each agent, upon receiving the UTIL message from each child, performs the same operations. Thus, these utilities will propagate up the pseudo-tree until they reach the root agent.

- *Value Propagation Phase.* This phase starts after the utility propagation (line 11) by having the root agent compute its optimal values $\check{\mathbf{x}}_i^k$ and $\hat{\mathbf{x}}_i^k$ for the relaxed DCOPs $\check{P}$ and $\hat{P}$, respectively (line 22). It then sends its values to all its neighbors in a VALUE message (line 23). When its child receives this message, it also compute its optimal values and sends them to all its neighbors (lines 31-33). Thus, these values propagate down the pseudo-tree until they reach the leaves, at which point every agent has chosen its respective values. In this phase, in preparation for the bounding phase, when each agent receives a VALUE message from its neighbor, it will also update the value of its neighbor in both its contexts $\check{\chi}_{a_i}^k$ and $\hat{\chi}_{a_i}^k$ (lines 24-26 and 29-30).

Bounding Phase. Once the relaxed DCOPs $\check{P}$ and $\hat{P}$ have been solved, the algorithm starts the bound propagation phase (line 12). This phase starts by having each leaf agent of the pseudo-tree $T^{\diamond}$ compute the lower and upper bounds LB_i^k and UB_i^k (lines 36-37). These bounds are sent to its parent in $T^{\diamond}$ (line 38). When its parent receives this message (line 35), it performs the same operations. The lower and upper bounds of the whole problem are determined when the bounds reach the root agent.

T-DBR Example Trace.

In order to elucidate the behavior of the proposed T-DBR algorithm we illustrate, in Figure 4.8, a running example of the algorithm during the first two D-LNS iterations. It uses the DCOP of Figure 4.1. The trees T^1 and T^2 are represented by bold solid lines (functions in E_{T^k}); all other functions are represented by dotted lines. The preserved variables in each iteration are shaded gray, and the functions in which they participate are represented by bold dotted lines. At each step, the resolution of the relaxed problems involves the functions represented by bold lines. We recall that while solving $\check{P}$ accounts for the function involving at least a destroyed variable, solving $\hat{P}$ focuses solely on the functions in E_{T^k} (i.e., involving exclusively destroyed variables). The nodes illustrating destroyed variables are labeled with red values representing $\check{\mathbf{x}}_i^k$; nodes representing preserved variables are labeled with black values representing $\check{\mathbf{x}}_i^{k-1}$. Each edge is labeled with a pair of values representing the utilities $\hat{f}^k(\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_j^k)$ (top) and $f(\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_j^k)$ (bot-

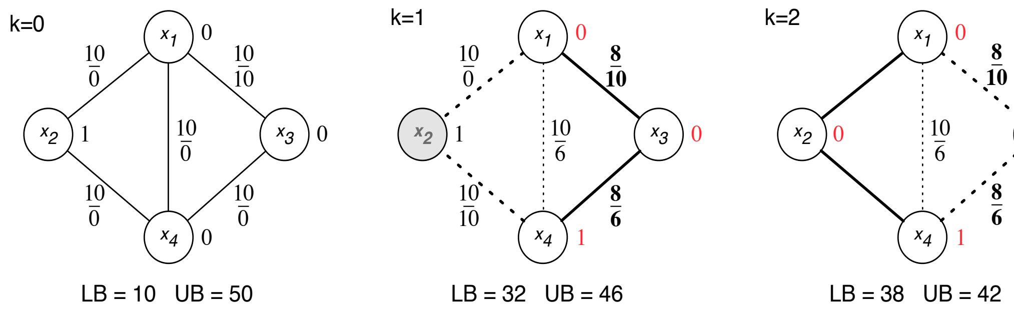


Figure 4.8: D-LNS with T-DBR example trace.

tom) of the corresponding functions. The lower and upper bounds of each iteration are shown below. When $k=0$, each agent randomly assigns a value to its variable, which results in a solution with utility

$$\mathbf{F}_g(\check{\mathbf{x}}^0) = f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_2^0) + f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_3^0) + f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_4^0) + f(\check{\mathbf{x}}_2^0, \check{\mathbf{x}}_4^0) + f(\check{\mathbf{x}}_3^0, \check{\mathbf{x}}_4^0) = 0 + 10 + 0 + 0 + 0 = 10$$

to get the lower bound. Moreover, solving $\hat{P}^0$ yields a solution $\hat{\mathbf{x}}^0$ with utility

$$\hat{F}^0(\hat{\mathbf{x}}^0) = \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_2^0) + \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_3^0) + \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_4^0) + \hat{f}^0(\hat{\mathbf{x}}_2^0, \hat{\mathbf{x}}_4^0) + \hat{f}^0(\hat{\mathbf{x}}_3^0, \hat{\mathbf{x}}_4^0) = 10+10+10+10+10 = 50,$$

which is the upper bound.

In the first iteration ($k=1$), the destroy phase preserves x_2, and thus $\check{\mathbf{x}}_2^1 = \check{\mathbf{x}}_2^0 = 1$. The algorithm then builds the spanning tree with the remaining variables choosing $f(x_1, x_3)$ and $f(x_3, x_4)$ as a tree edges. Solving $\check{P}^1$ yields solution $\check{\mathbf{x}}^1$ with utility

$$\check{F}^1(\check{\mathbf{x}}^1) = f(\check{\mathbf{x}}_1^1, \check{\mathbf{x}}_3^1) + f(\check{\mathbf{x}}_3^1, \check{\mathbf{x}}_4^1) + f(\check{\mathbf{x}}_1^1, \check{\mathbf{x}}_2^1) + f(\check{\mathbf{x}}_2^1, \check{\mathbf{x}}_4^1) = 10 + 6 + 0 + 10 = 26,$$

which results in a lower bound

$$\mathbf{F}_g(\check{\mathbf{x}}^1) = \check{F}^1(\check{\mathbf{x}}^1) + f(\check{\mathbf{x}}_1^1, \check{\mathbf{x}}_4^1) = 26 + 6 = 32.$$

Solving $\hat{P}^1$ yields solution $\hat{\mathbf{x}}^1$ with utility

$$\hat{F}^1(\hat{\mathbf{x}}^1) = \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_2^1) + \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_3^1) + \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_4^1) + \hat{f}^1(\hat{\mathbf{x}}_2^1, \hat{\mathbf{x}}_4^1) + \hat{f}^1(\hat{\mathbf{x}}_3^1, \hat{\mathbf{x}}_4^1) = 10+8+10+10+8 = 46,$$

which is the current upper bound. Recall that the values for the functions in $\tilde{E}^k$, are computed as $\frac{\tilde{F}^k(\mathbf{x})}{|\tilde{E}^k|} = \frac{16}{2} = 8$.

Finally, in the second iteration ($k=2$), the destroy phase retains x_3 assigning it its value in the previous iteration $\check{\mathbf{x}}_3^2 = \check{\mathbf{x}}_3^1 = 0$, and the repair phase builds the new spanning tree with the remaining variables. Solving $\check{P}^2$ and $\hat{P}^2$ yields solutions $\check{\mathbf{x}}^3$ and $\hat{\mathbf{x}}^3$, respectively, with utilities

$$\check{F}^2(\check{\mathbf{x}}^2) = 10 + 6 + 10 + 6 = 32,$$

which results in a lower bound

$$\mathbf{F}_g(\check{\mathbf{x}}^2) = 32 + 6 = 38,$$

and an upper bound

$$\hat{F}^2(\hat{\mathbf{x}}^2) = 8 + 8 + 10 + 8 + 8 = 42.$$

4.3.3 Theoretical Analysis

In this section, we provide a theoretical discussion on the bounds provided by our D-LNS framework with the DBR repair algorithm. These results describe how D-LNS with DBR provides quality guarantees. Additionally, we discuss the complexity analysis for the network load and the messages size of T-DBR, as well as the agent's complexity.

Theorem 8. *For each LN^k,*

$$\mathbf{F}_g(\check{\mathbf{x}}^k) \leq \mathbf{F}_g(\mathbf{x}^*).$$

Proof. The result follows from that $\check{\mathbf{x}}^k$ is an optimal solution of the relaxed problem $\check{P}$ whose functions are a subset of $\mathbf{F}$. □

Lemma 3. *For each k,*

$$\sum_{f \in \tilde{E}^k} \hat{f}(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \geq \sum_{f \in \tilde{E}^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*),$$

where $\hat{\mathbf{x}}_i^k$ is the value assignment to variable x_i when solving the relaxed DCOP $\hat{P}$ and $\mathbf{x}_i^*$ is the value assignment to variable x_i when solving the original DCOP P.

Proof. It follows that:

$$\begin{aligned}
\sum_{f \in \tilde{E}^k} \hat{f}(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) &\geq \sum_{f \in \tilde{E}^k} \max\left\{\frac{\tilde{F}^k}{|\tilde{E}^k|}, \hat{f}(\hat{\mathbf{x}}_i^{k-1}, \hat{\mathbf{x}}_j^{k-1})\right\} && \text{(by Definition of } \hat{f}\text{)}\\
&\geq \sum_{f \in \tilde{E}^k} \frac{\tilde{F}^k}{|\tilde{E}^k|}\\
&\geq \tilde{F}^k\\
&= \sum_{f \in \tilde{E}^k} f(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) && \text{(by Definition of } \tilde{F}^k\text{)}\\
&\geq \sum_{f \in \tilde{E}^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*).
\end{aligned}$$

The last step follows from that, in each iteration k, the functions associated with the tree edges in $\tilde{E}^k$ are solved optimally. Since their cost is maximized it is also greater than the optimal one. □

Lemma 4. *For each k,*

$$\sum_{f \in \Theta^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \geq \sum_{f \in \Theta^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*),$$

where $\Theta^k = \{f \in \mathcal{F} \mid \Gamma_f^k \neq \emptyset\}$ is the set of functions that have been chosen as edges of the relaxation graph in a previous iteration.

Proof. We prove it by induction on the iteration k. For ease of explanation we provide an illustration (on the bottom right of the page) of the set of relevant edges optimized in successive iterations.

For $k = 0$, then $\Theta^0 = \emptyset$ and, thus, the statement vacuously holds. Assume the claim holds up to iteration $k - 1$. For iteration k it follows that,

$$\begin{aligned}
&\sum_{f \in \Theta^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&= \sum_{f \in \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&= \sum_{f \in \Theta^{k\text{-}1} \setminus \tilde{E}^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&= \sum_{f \in \Theta^{k\text{-}1} \setminus \tilde{E}^k} \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1}) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \max\left\{\hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k), \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1})\right\} + \sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k)
\end{aligned}$$

(by definition of $\hat{f}^k$)

Since,

$$\begin{aligned}
\sum_{f \in \Theta^{k\text{-}1} \setminus \tilde{E}^k} \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1}) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \max\left\{\hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k), \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1})\right\} &\geq \sum_{f \in \Theta^{k\text{-}1}} \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1}); \\
\sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \max\left\{\hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k), \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1})\right\} &\geq \sum_{f \in \tilde{E}^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k).
\end{aligned}$$

It follows:

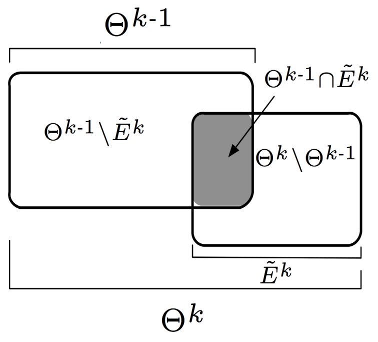
$$\begin{aligned}
&\sum_{f \in \Theta^{k\text{-}1} \setminus \tilde{E}^k} \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1}) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \max\left\{\hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k), \hat{f}^{k\text{-}1}(\hat{\mathbf{x}}_i^{k\text{-}1}, \hat{\mathbf{x}}_j^{k\text{-}1})\right\} + \sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&\geq \sum_{f \in \Theta^{k\text{-}1} \setminus \tilde{E}^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*) + \sum_{f \in \Theta^{k\text{-}1} \cap \tilde{E}^k} \max\left\{f(\mathbf{x}_i^*, \mathbf{x}_j^*), f(\mathbf{x}_i^*, \mathbf{x}_j^*))\right\} + \sum_{f \in \Theta^k \setminus \Theta^{k\text{-}1}} f(\mathbf{x}_i^*, \mathbf{x}_j^*)
\end{aligned}$$

(by Lemma 3 and induction assumption)

$$\geq \sum_{f \in \Theta^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*).$$

□

Lemma 4 ensures that the utility associated to the functions optimized in the relaxed problems $\hat{P}$, up to iteration k, is an upper bound for the evaluation of the same set of functions, evaluated under the optimal solution for P. The above proof relies on the observation that the functions in Θ^k includes exclusively those ones associated with the optimization of problems $\hat{P}^\ell$, with $\ell \leq k$, and that the functions over which the optimization process operates multiple times, are evaluated with their maximal value observed so far.


$\Theta^{k\text{-}1}$
$\Theta^{k\text{-}1} \cap \tilde{E}^k$
$\Theta^{k\text{-}1} \setminus \tilde{E}^k$
$\Theta^k \setminus \Theta^{k\text{-}1}$
$\tilde{E}^k$
Θ^k

Theorem 9. *For each LN^k, $\hat{F}^k(\hat{\mathbf{x}}_k) \geq \mathbf{F}_g(\mathbf{x}^*)$.*

Proof. By definition of $\hat{F}^k(\mathbf{x})$, it follows that,

$$\begin{aligned}
\hat{F}^k(\mathbf{x}) &= \textstyle\sum_{f\in\mathbf{F}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&= \sum_{f\in\Theta^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f\notin\Theta^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
&= \sum_{f\in\Theta^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f\notin\Theta^k} \max_{d_i,d_j} f(d_i, d_j) && \text{(by definition of } \hat{f}^k\text{)} \\
&\geq \sum_{f\in\Theta^k} f(x_i^*, x_j^*) + \sum_{f\notin\Theta^k} f(x_i^*, x_j^*) && \text{(by Lemma 4)} \\
&= \mathbf{F}_g(\mathbf{x}^*)
\end{aligned}$$

which concludes the proof. □

Corollary 4. *An approximation ratio for the problem is*

$$\rho = \frac{\min_k \hat{F}^k(\hat{\mathbf{x}}^k)}{\max_k \mathbf{F}_g(\check{\mathbf{x}}^k)} \geq \frac{\mathbf{F}_g(\mathbf{x}^*)}{\max_k \mathbf{F}_g(\check{\mathbf{x}}^k)}$$

Proof. This result follows from $\max_k \mathbf{F}_g(\check{\mathbf{x}}^k) \leq \mathbf{F}_g(\mathbf{x}^*)$ (Theorem 8) and $\min_k \hat{F}^k(\hat{\mathbf{x}}^k) \geq \mathbf{F}_g(\mathbf{x}^*)$ (Theorem 9). □

Theorem 10. *In each iteration, T-DBR requires $O(|\mathbf{F}|)$ number of messages of size $O(d)$, where $d = \max_{a_i\in\mathbf{A}} |D_i|$.*

Proof. The number of messages required at each iteration is bounded by the *Value Propagation Phase* of Algorithm 2, where each agent sends a message to each of its neighbors (lines 23 and 33). In contrast all other phases use up to $|A|$ messages (which are reticulated from the leaves to the root of the pseudo-tree and vice-versa). The size of the messages is bounded by the *Utility Propagation Phase*, where each agent (excluding the root agent) sends a message containing a value for each element of its domain (line 20). All other messages exchanged contain two values (lines 23, 33, and 38). Thus the maximum size of the messages exchanged at each iteration is at most d. □

Theorem 11. *In each iteration, the number of constraint checks of each T-DBR agent is $O(d^2)$, where $d = \max_{a_i\in\mathbf{A}} |D_i|$.*

Proof. The number of constraint checks, performed by each agent in each iteration, is bounded by the operations performed during the Util-Propagation Phase. In this phase, each agent (except the root agent) computes the lower and upper bound utilities for each values of its variable $\mathbf{x}_i$ and its parent's variable $\mathbf{x}_{P^k_{a_i}}$ (lines 16–17). □

4.3.4 Related Work

Due to the vast amount of resources and/or communication required to solve DCOP optimally, an increasing amount of effort has been put by researchers to propose several incomplete DCOP solutions. These can be grouped into three categories: *search-based*, *inference-based*, and *region-optimal* algorithms, based on the processing technique adopted to explore the solution space. Incomplete search-based algorithms (e.g., DSA [126], MGM [72]) are based on the use of incomplete search techniques to explore the space of possible solutions. Incomplete inference-based algorithms (e.g., Max-Sum [31]) use solutions inspired from dynamic programming and belief propagation techniques. Region-optimal algorithms allow us to specify regions with a maximum size of k agents or t hops from each agent, and they optimally solve the subproblem within each region. Unfortunately, several local search algorithms (e.g., DSA [126], MGM [72]) and local inference algorithms (e.g., Max-Sum [31]) do not provide guarantees on the quality of the solutions found. More recent developments, such as region-optimal algorithms [91, 117], Bounded Max-Sum [104], and DaC algorithms [116, 54] alleviate this limitation. In region-optimal algorithms, solution quality bounds are provided as a function of k or t. Bounded Max-Sum is an extension of Max-Sum, which solves optimally an acyclic version of the DCOP graph, bounding its solution quality as a function of the edges removed from the cyclic graph. DaC-based algorithms use Lagrangian decomposition techniques to solve agent subproblems sub-optimally. Good quality assessments are essential for sub-optimal solutions.

Aside from these incomplete algorithms, researchers have also developed extensions to complete algorithms that trade solution quality for faster runtime. For example, complete search algorithms have mechanisms that allow users to specify absolute or relative error bounds [81, 122]. Researchers have also worked on non-iterative versions of inference-based incomplete DCOP algorithms, with and without quality guarantees [104, 86, 95]. Such methods are, however, unable to refine the initial solution returned. Finally, the algorithm that is the most similar to ours is LS-DPOP [98], which operates on a pseudo-tree performing a local search. However, unlike D-LNS, LS-DPOP operates only in a single iteration, does not change its neighborhood, and does not provide quality guarantees.

4.3.5 Experimental Evaluation

We evaluate the D-LNS framework against state-of-the-art incomplete DCOP algorithms, with and without quality guarantees, where we choose representative *search*-, *inference*-, and *region optimal*-based solution approaches. We select Distributed Stochastic Algorithm (DSA) as a representative of an incomplete search-based DCOP algorithm; Max-Sum (MS), and Bounded Max-Sum (BMS), as representative of inference-based DCOP algorithms, and k- and t-optimal algorithms (KOPT, and TOPT), as representative of region optimal-based DCOP methods. All algorithms are selected based on their performance and popularity. We run the algorithms using the following implementations: We use the FRODO framework [68] to run MS, and DSA,[5] we use the authors' code of BMS [104], and the DALO framework [58] for KOPT and TOPT. We run all algorithms using their default parameters, thus the number of iterations for MaxSum, DSA, and K-,T-OPT is set to 500, 200, and 100, respectively. We use DSA-B with $p = 0.6$. We systematically evaluate the runtime, solution quality and network load of the algorithms on binary constraint networks with *random*, *scale-free*, and *grid* topologies, and we evaluate the ability of D-LNS to exploit domain knowledge over *distributed meeting scheduling problems*.

[5] As a technical note, we implement DSA-B which required minimal changes from DSA-C, and readily available on FRODO.

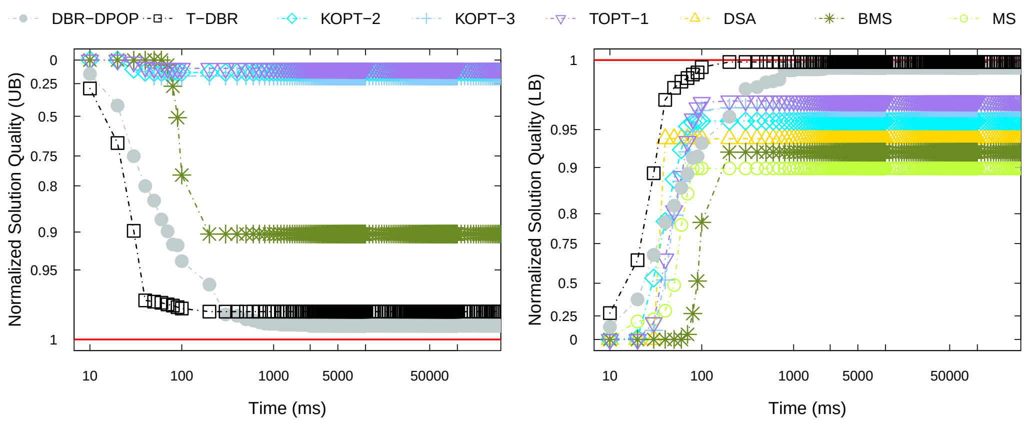


Figure 4.9: Normalized solution quality for the upper bounds and lower bounds, on regular grids at varying of the maximum time allotted to the algorithms.

The instances for each topology are generated as follows:

- ***Random:*** We create an n-node network, whose density p_1 produces $\lfloor n\,(n-1)\,p_1 \rfloor$ edges in total. We do not bound the tree-width, which is based on the underlying graph.
- ***Scale-free:*** We create an n-node network based on the Barabasi-Albert model [5]. Starting from a connected 2-node network, we repeatedly add a new node, randomly connecting it to two existing nodes. In turn, these two nodes are selected with probabilities that are proportional to the numbers of their connected edges. The total number of edges is $2\,(n-2)+1$.
- ***Grid:*** We create an n-node network arranged in a rectangular grid, where internal nodes are connected to four neighboring nodes and nodes on the edges (resp. corners) are connected to two (resp. three) neighbors.

We generate 50 instances for each topology, ensuring that the underlying graph is connected. The utility functions are generated using random integer costs in $[0, 100]$. We set as default parameters, $|\mathcal{A}| = 20$, $|D_i| = 10$ for all variables, and $p_1 = 0.5$ for random networks. We use a random destroy strategy for the D-LNS algorithms. Algorithms runtimes are measured using the *simulated runtime* metric [111], and we impose a timeout of 300s. Results are averaged over all instances and are statistically significant[6] with p-values < 0.0001. The experiment are performed on an Intel i7 Quadcore 3.3GHz machine with 4GB of RAM.

Figure 4.9, 4.10, and 4.11 illustrates the convergence results (normalized upper and lower bounds) for, respectively, regular grids , random graphs, and scale-free networks in increasing amounts of maximum time allowed to the algorithms to complete. Figure 4.12, 4.13, and 4.14 illustrates the convergence results (normalized upper and lower bounds) for, respectively, regular grids , random graphs, and scale-free networks in increasing amounts of maximum network load allowed to the algorithms to complete. A value of 0 (1), means worst (best) lower or upper bound w.r.t. the lower or upper bound reported within

[6] t-test performed with null hypothesis: DLNS-based algorithms find solution with better bounds than non-DLNS based ones.

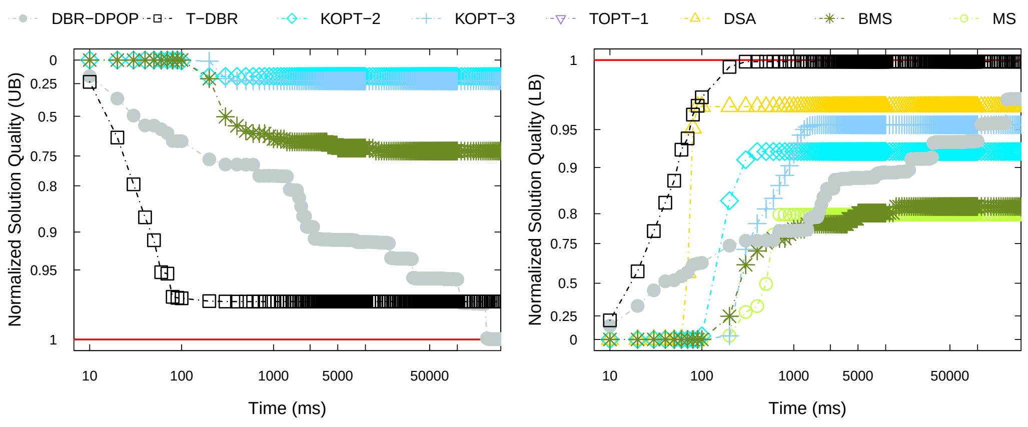


Figure 4.10: Normalized solution quality for the upper bounds and lower bounds, on random graphs at varying of the maximum time allotted to the algorithms.

the pool of algorithms examined. All plots are in log-scale. These results show that the D-LNS-based algorithms not only converge to better solutions, but converge to them faster, and with lower network load requirements. In addition, they provide tighter upper bounds, and thus find better approximation ratios compared to the other algorithms. The figures reporting the upper bounds do not illustrate MS and DSA, as they do not provide bounded solutions. TOPT-1 timed-out for all instances on random and scale-free networks. D-LNS with DPOP-DBR is slower than D-LNS with T-DBR, and it reaches a timeout for the scale-free networks. This is due to the fact that the complexity of the former repair phase is exponential in the induced width of the relaxed constraint graph, and scale-free exhibit higher induced widths than grids and random network instances. In contrast, D-LNS with T-DBR does not encounter such limitations. The main reason behind fast convergence to good solutions of the D-LNS algorithms is that, on average, about half of the agents are destroyed at each iteration, thus reducing the search space significantly.

Table 4.2 reports the solution qualities of the different algorithms on random networks. We report the approximation ratio ρ and the ratio ϵ of the best quality found by all algorithms versus its quality. Best approximation ratios and quality ratios are shown in bold. The results show that D-LNS with DBR-DPOP finds better approximation ratios ρ than those of the competing algorithms. However, it fails to solve problems bigger than 20 agents. In contrast, D-LNS with T-DBR can scale to large problems better than other algorithms. Similarly to the trends observed in the previous experiment, D-LNS with T-DBR finds better solutions w.r.t. the other algorithms (i.e., better quality ratios ϵ and better approximation ratios ρ for $|\mathbf{A}| > 20$).

Distributed Meeting Scheduling. Many real-world problems model require the use of hard constraints, to avoid considering infeasible solutions (see, e.g., `http://www.csplib.org`). We also evaluate the ability of our D-LNS framework to exploit problem structure, exhibited in presence of domain-dependent knowledge and hard constraints, and test its behavior on *distributed meeting scheduling problems*. In such problems, one wishes to schedule a set of events within a time range. We use the *time slots as variable* formulation [72], where events are modeled as decision variables. Meeting participants can attend different meetings, and have time preferences that are taken into account in the problem formulation. Each variable can take on a value from the time slot range in $[0, 100]$, that is sufficiently early to schedule the

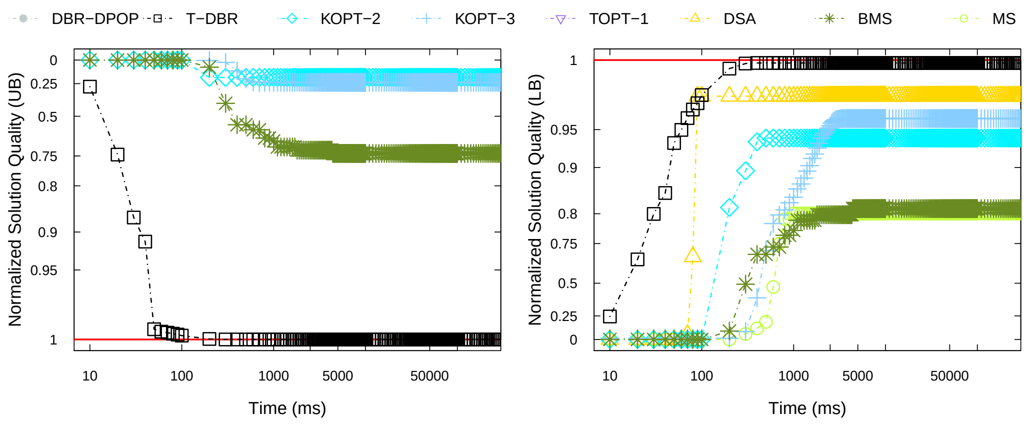


Figure 4.11: Normalized solution quality for the upper bounds and lower bounds, on scale-free networks at varying of the maximum time allotted to the algorithms.

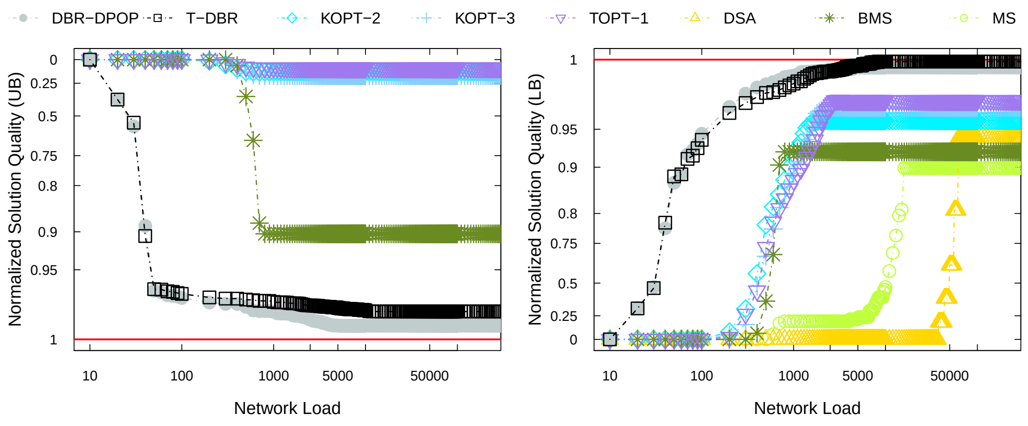


Figure 4.12: Normalized solution quality for the upper bounds and lower bounds, on regular grids at varying of the maximum network load allotted to the algorithms.

$\|\mathbf{A}\|$	DBR-DPOP		T-DBR		BMS		KOPT2		KOPT3		TOPT1		MaxSum	DSA
	ρ	ϵ	ρ	ϵ	ρ	ϵ	ρ	ϵ	ρ	ϵ	ρ	ϵ	ϵ	ϵ
10	**1.055**	0.997	1.149	**0.999**	1.872	0.824	4.333	0.935	3.500	0.969	6.000	0.989	0.779	0.941
20	**1.278**	0.977	1.311	**0.999**	2.302	0.819	7.666	0.923	6.000	0.954	–		0.797	0.971
50	–		**1.539**	**0.995**	3.001	0.849	17.66	0.900	13.50	0.907	–		0.832	0.988
100	–		**1.669**	**1.000**	2.797	0.871	34.33	0.892	26.00	0.897	–		0.866	0.975
200	–		**1.759**	**1.000**	2.878	0.897	67.66	0.898	–		–		–	0.973

Table 4.2: Experimental results on *random* networks.

required participant for the required amount of time. The problem requires that no meetings sharing some participants overlap. We generate the underlying constraint network using the random network model de-

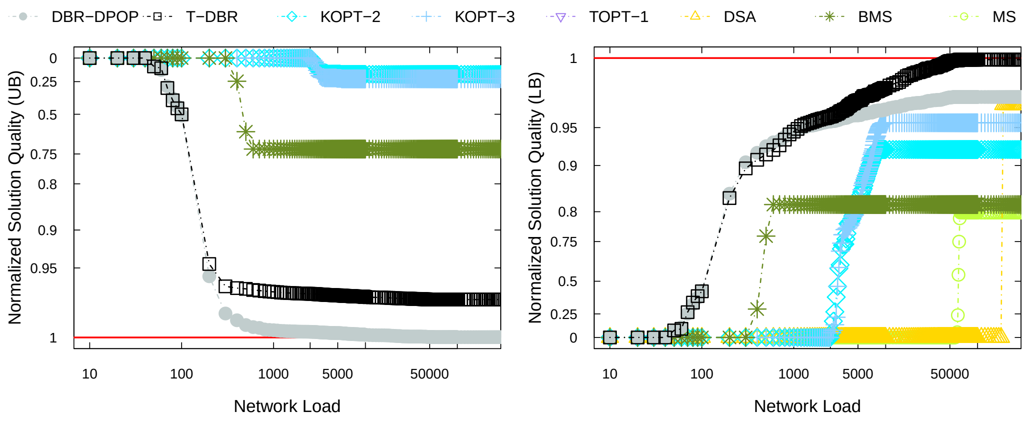


Figure 4.13: Normalized solution quality for the upper bounds and lower bounds, on random graphs at varying of the maximum network load allotted to the algorithms.

Meetings:	20		50		100	
	% SAT	TF (ms)	% SAT	TF (ms)	% SAT	TF(ms)
DK destroy	**80.05**	**78**	**54.11**	**342**	**31.20**	**718**
RN destroy	12.45	648	1.00	52207	0.00	–
KOPT3	4.30	110367	0.00	–	–	–

Table 4.3: Experimental results on *meeting scheduling*.

scribed earlier. We compare the repair phase T-DBR with both *random* (RN) destroy and *domain-specific knowledge* (DK) destroy methods. The latter destroys the set of variables in overlapping meetings. Table 4.3 reports the percentage of satisfied instances reported (% SAT) and the time needed to find the first satisfiable solution (TF), averaged over 50 runs. The domain-specific destroy method has a clear advantage over the random one, being able to effectively exploit domain knowledge. All other local search algorithm failed to report satisfiable solutions for any of the problems—only KOPT3 was able to find some satisfiable solutions for 20 meetings.

4.4 Summary

This chapter introduced two DCOP solving strategies, *Branch Consistency* (BrC) and *Distributed Large Neighborhood Search* (D-LNS), which adapt centralized reasoning techniques exploiting the structure of DCOPs during the problem resolution phase, to enhance the DCOP resolution efficiency.

BrC is a type of consistency that applies to paths in pseudo-trees, and it is aimed to prune the search space and to reduce the size of the messages exchanged among agents by actively exploiting the hard constraints of the problem. Our experimental results show that when applied to DPOP such form of consistency enforces a pruning that is more effective than that enforced by Arc Consistency. We experimentally show that the resulting algorithm, called BrC-DPOP, can prune as much as a version of H-DPOP

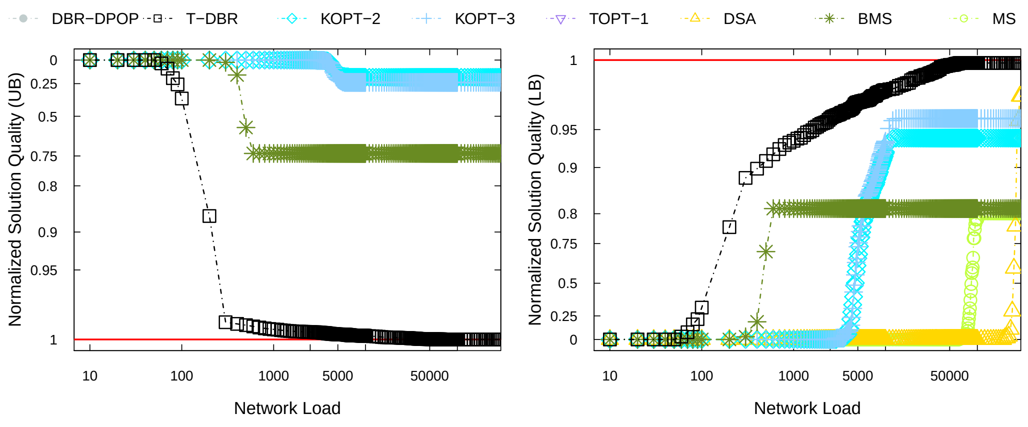


Figure 4.14: Normalized solution quality for the upper bounds and lower bounds, on scale-free networks at varying of the maximum network load allotted to the algorithms.

that limits its knowledge to the same amount as BrC-DPOP in a much smaller amount of time. We also show that it can scale to larger problems than DPOP and H-DPOP. Therefore, these results confirm the strengths of this approach, leading to enhanced efficiency and scalability.

While BrC has been applied to a complete algorithm, D-LNS is a framework that can be used to find quality-bounded approximated solutions in DCOPs. Thus, it defines a DCOP incomplete algorithmic framework. D-LNS is composed of a destroy phase, which selects a large neighborhood to search, and a repair phase, which performs the search over the selected neighborhood. We introduce two novel distributed repair phases, DBR-DPOP and T-DBR, built within the D-LNS framework, and characterized by low network usage; additionally, T-DBR provides a low computational complexity per agent. Our experimental results show that, using to its ability to exploit large neighbors, the D-LNS algorithms not only converge to better solutions, compared to incomplete DCOP algorithms that are representative of search-based, inference-based, and region-optimal-based approaches, but converge to them faster, and with low network load requirements. The proposed results are significant—the anytime property, the ability of refining online quality guarantees, and the ability to exploit domain-dependent structure, makes D-LNS-based algorithms good candidates to solve a wide class of DCOP problems.

These two works, explore two orthogonal mechanisms which adapt centralized reasoning to the DCOP resolution process, and were applied to produce a complete and an incomplete DCOP solving approach. Therefore, these results validate the hypothesis that centralized reasoning can be adapted to exploit the structure of DCOPs during problem solving to enhance the DCOP solving efficiency.

5 Exploiting the use of Accelerated Hardware in DCOP resolution

Typical Distributed Constraint Optimization problems are characterized by complex dynamics and interactions among a large number of agents, which translate into hard combinatorial problems, posing significant challenges from a computational point of view. To deal with such computational burden, and in addition to the techniques discussed in the previous chapters, this chapter studies how to exploit a novel class of massively parallel platforms that are based on the *Single Instruction Multiple Thread* (SIMT) paradigm, and widely used in modern General Purpose Graphic Processing Units (GPGPU)s. The wide availability of GPGPUs, and their contained costs, stimulated spread interest across several research communities. Thus, in this chapter we propose the design and implementation of inference-based and sampling-based algorithms which exploits GPGPU parallel architectures to speed up the resolution of DCOPs.

The structure exploited by *Dynamic Programming* (DP)-based approaches in constructing solutions makes it suitable to harness the SIMT paradigm. Thus, we proposed a DP-based algorithm that makes use of parallel computation using GPGPUs to solve DCOPs [34]. Our results show significant improvements in performance and scalability over other state-of-the-art DP-based solutions.

The explicit separation between the DCOP resolution process and the centralized agent problem, enabled by our MVA DCOP decomposition (see Chapter 3), enables agents to solve their local problem trough a variety of techniques. Motivated by the high complexity of the agent local problem, we proposed the use of hierarchical parallel models, where each agent can *(i)* solve its local problem independently from those of other agents, and *(ii)* parallelize the computations within its own local problem. Thus, in this chapter we introduce a framework to solve independent local problems, in parallel, using sampling-based algorithms, harnessing the multitude of computational units offered by GPGPUs. This approach led to significant improvements in the runtime of the algorithm resolution. Therefore, these results validate our hypothesis that one can exploit highly parallel computational models to enhance current DCOP solution techniques through the design of algorithmic approaches that take advantage of such novel hardware architectures.

This chapter is structured as follows: The next section introduces the motivations for the adoption of accelerated hardware to solve DCOPs. In section 5.2 we describe a design and implementation of a DP-based algorithm that exploits parallel computation using GPGPUs. section 5.3 introduces a GPGPU-based Monte Carlo Markov Chain (MCMC) framework to solve the agents' independent subproblems exposed via the MVA DCOP decomposition. Therein, we detail the GPGPU implementations of Gibbs and Metropolis-Hasting, two MCMC sampling algorithms. In each of the latter two sections we report the description of the frameworks, a theoretical analysis on the relevant properties exposed by the two solving techniques, and the experimental results. Finally, section 5.4 concludes the Chapter.

5.1 Motivations

5.1.1 DP-based Algorithms

The importance of *Dynamic Programming* (DP)-based approaches arises in several optimization fields including constraint programming [2, 7]. For example, several *propagators* adopt DP-based techniques to establish constraint consistency; for instance,

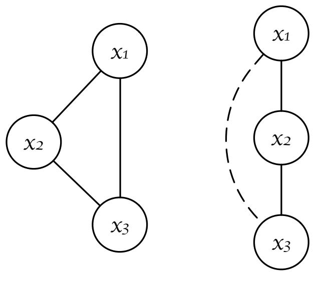
1. the *knapsack* constraint propagator proposed by Trick applies DP techniques to establish arc consistency on the constraint [114];
2. the propagator for the *regular* constraint establishes arc consistency using a specific digraph representation of the DFA, which has similarities to dynamic programming [94];
3. the *context free grammar* constraint makes use of a propagator based on the CYK parser that uses DP to enforce generalized arc consistency [102].

The importance of DP arises also in several declarative constraint programming languages. For instance, the language *PICAT* [129, 127] makes use of table constraints to implement DP-based resolution approaches [130, 128]

While DP approaches may not always be appropriate to solve (D)COPs, as their time and space requirements may be prohibitive, they may be very effective in problems with particular structures, such as problems where their underlying constraint graphs have small induced widths or distributed problems where the number of messages is crucial for performance, despite the size of the messages. The structure used by DP-based approaches in constructing solutions makes it suitable to exploit a novel class of massively parallel platforms that are based on the Single Instruction Multiple Thread paradigm—where multiple threads may concurrently operate on different data, but are all executing the same instruction at the same time. The SIMT-based paradigm is widely used in modern Graphical Processing Units for general purpose parallel computing. We have applied such form of parallelism to enhance the resolution efficiency of DP-based algorithms to solve COPs and DCOPs, resulting in a new framework, called *GPU-DBE* and introduced in section 5.2. Crucially, agents within the GPU-DBE framework can effectively make use of the power harnessed by the GPGPUs, resulting in enhanced running time and scalability.

5.1.2 Exploiting MVA Hierarchical Parallelism

Exploiting the use of MVA-based decompositions for DCOPs, we introduce a general framework, called *Distributed MCMC (DMCMC)* which is based on the *Distributed Pseudo-tree Optimization Procedure* (DPOP) algorithm [96] to allow each agent to solve its local sub-problem using *Markov Chain Monte*



(a) (b)

for i < j

xi	*xj*	Utilities
0	*0*	*5*
0	*1*	*8*
1	*0*	*20*
1	*1*	*3*

(c)

x1	*x2*	Utilities	*x3*
0	*0*	*max(5+5, 8+8) = 16*	*1*
0	*1*	*max(5+20, 8+3) = 25*	*0*
1	*0*	*max(20+5, 3+8) = 25*	*0*
1	*1*	*max(20+20, 3+3) = 40*	*0*

x1	Utilities	*x2*	*x3*
0	*max(5+16, 8+25) = 33*	*1*	*0*
1	*max(20+25, 3+40) = 45*	*0*	*0*

(d)

Figure 5.1: Example (D)COP (a-c) and *UTIL* phase computations in DPOP (d).

Carlo (MCMC) sampling algorithms. The data independence property exposed by such sampling algorithms, makes this framework suitable to effectively exploit SIMT-based parallelism, and we thus use GPGPU hardware to parallelize and speed up the sampling process. In section 5.3 we demonstrate the generality of this framework using two commonly used MCMC algorithms, the Gibbs [38] and Metropolis-Hastings [53, 77] algorithms. Our experiments show that our framework is able to find near-optimal solutions up to two orders of magnitude faster than MGM and MGM2 (two local search DCOP algorithms).

5.2 Accelerating DPOP and BE resolution on GPGPUs

This section proposes a design and implementation of the GPU-based (Distributed) Bucket Elimination framework (GPU-DBE), a DP-based algorithm that exploits parallel computation using GPGPUs to solve (D)COPs. Our proposal aims at employing GPGPU hardware to speed up the inference process of DP-based methods, representing an alternative way to enhance the performance of DP-based constraint optimization approaches. The underlying structure exploited by DP-based approaches in constructing solutions allows us to fully utilize the power of the GPGPU hardware, exploring in parallel a large number of operations performed during the inference steps. Specifically, we focus on the parallelization of the Bucket Elimination (BE) procedure [25], which is a DP-based algorithm for solving COPs, and of the DPOP algorithm, which can be seen as a distributed version of BE, where agents exchange newly introduced utility functions via messages. The effect of exploiting such type of accelerated-solutions provides significant advantages in terms of runtime and scalability, resulting in speedups up to two orders of magnitude, with respect to an optimized sequential version of the same solver.

Throughout the section, we will use the example DCOP shown in Figure 5.1, to describe the behavior of the inference process within BE and DPOP. Figure 5.1(a) shows the constraint graph of a simple COP with three variables, x_1, x_2, and x_3. The domain of each variable is the set $\{0, 1\}$. Figure 5.1(c) describes the utility functions of the COP.

5.2.1 Notation and Definitions

We recall that θ denote a (D)COP *solution*, and introduce the following definitions:

Definition 14 (Projection)**.** *The* projection *of a utility function f_i on a set of variables $\mathbf{V} \subseteq \mathbf{x}^i$ is a new utility function $f_{i|\mathbf{V}} : \mathbf{V} \to \mathbb{R}^+ \cup \{-\infty\}$, such that for each possible assignment $\theta \in \bigtimes_{x_j \in \mathbf{V}} D_j$, $f_{i|\mathbf{V}}(\theta) = \max\limits_{\sigma \in \Sigma, \sigma_{\mathbf{V}} = \theta} f_i(\sigma_{\mathbf{x}^i})$.*

In other words, $f_{i|\mathbf{V}}$ is constructed from the tuples of f_i, removing the values of the variable that do not appear in $\mathbf{V}$ and removing duplicate values by keeping the maximum utility of the original tuples in f_i.

Definition 15 (Concatenation)**.** *Let us consider two assignments θ', defined for variables V, and θ'', defined for variables W, such that for each $x \in V \cap W$ we have that $\theta'(x) = \theta''(x)$. Their* concatenation *is an assignment $\theta' \cdot \theta''$ defined for $V \cup W$, such as for each $x \in V$ (respectively $x \in W$) we have that $\theta' \cdot \theta''(x) = \theta'(x)$ (respectively $\theta' \cdot \theta''(x) = \theta''(x)$).*

We define two operations on utility functions:

- The *aggregation* of two functions f_i and f_j, is a function $f_i + f_j : \mathbf{x}^i \cup \mathbf{x}^j \to \mathbb{R}^+ \cup \{-\infty\}$, such that $\forall \theta' \in \bigtimes_{x_k \in \mathbf{x}^i} D_k$ and $\forall \theta'' \in \bigtimes_{x_k \in \mathbf{x}^j} D_k$, if $\theta' \cdot \theta''$ is defined, then we have that $(f_i + f_j)(\theta' \cdot \theta'') = f_i(\theta') + f_j(\theta'')$.
- *Projecting out* a variable $x_j \in \mathbf{x}^i$ from a function f_i, denoted as $\pi_{-x_j}(f_i)$, produces a new function with scope $\mathbf{x}^i \setminus \{x_j\}$, and defined as the projection of f_i on $\mathbf{x}^i \setminus \{x_j\}$, i.e., $\pi_{-x_j}(f_i) = f_{i|\mathbf{x}^i \setminus \{x_j\}}$.

Bucket Elimination (BE)

BE [25, 26] is a dynamic programming based procedure that can be used to solve COPs. Algorithm 2 illustrates its pseudocode. Given a COP $(\mathbf{X}, \mathbf{D}, \mathbf{C})$ and an ordering $o = \langle x_1, \dots, x_n \rangle$ on the variables in $\mathbf{X}$, we say that a variable x_i has a higher *priority* with respect to variable x_j if x_i appears after x_j in o. BE operates from the highest to lowest priority variable. When operating on variable x_i, it creates a bucket B_i, which is the set of all utility functions that involve x_i as the highest priority variable in their scope (line 2). The algorithm then computes a new utility function $\hat{f}_i$ by aggregating the functions in B_i and projecting out x_i (line 3). Thus, x_i can be removed from the set of variables $\mathbf{X}$ to be processed (line 4) and the new function $\hat{f}_i$ replaces in $\mathbf{C}$ all the utility functions that appear in B_i (line 5). In our example, BE operates, in order, on the variables x_3, x_2, and x_1. When x_3 is processed, the bucket B_3 is $\{f_{13}, f_{23}\}$, and the $\hat{f}_3$ utility function is shown in Figure 5.1(d) top. The rightmost column shows the values for x_3 after its projection. BE updates the sets $\mathbf{X} = \{x_1, x_2\}$ and $\mathbf{C} = \{f_{12}, \hat{f}_3\}$. When x_2 is processed, $B_2 = \{f_{12}, \hat{f}_3\}$ and $\hat{f}_2$ is shown in Figure 5.1(d) bottom. Thus, $\mathbf{X} = \{x_1\}$ and $\mathbf{C} = \{\hat{f}_2\}$. Lastly, the algorithm processes x_1, sets $B_1 = \{\hat{f}_2\}$, and $\hat{f}_1$ contains one value combination $\sigma^* = \langle 1, 0, 0 \rangle$, which corresponds to an optimal solution to the problem.

The complexity of the algorithm is bounded by the time needed to process a bucket (line 3), which is exponential in number of variables in the bucket.

Dynamic Programming Optimization Protocol (DPOP)

DPOP [96] is a dynamic programming based DCOP algorithm, introduced in section 2.1.4. Let us analyze some details of the algorithm which were not discussed earlier, and that we will use to observe the similarities between DPOP and BE.

Algorithm 2: BE

```
39 for $i \leftarrow n$ downto $1$ do
40     $B_i = \{f_j \in \mathbf{C} \mid x_i \in \mathbf{x}^j \wedge i = \max\{k \mid x_k \in \mathbf{x}^j\}\}$;
41     $\hat{f}_i = \pi_{-x_i}\left(\sum_{f_j \in B_i} f_j\right)$;
42     $\mathbf{X} = \mathbf{X} \setminus \{x_i\}$;
43     $\mathbf{C} = (\mathbf{C} \cup \{\hat{f}_i\}) \setminus B_i$;
```

In the UTIL propagation, each DPOP-agent, starting from the leaves of the pseudo-tree, computes the optimal sum of utilities in its subtree for each value combination of variables in its separator. The agent does so by aggregating the utilities of its functions with the variables in its separator and the utilities in the *UTIL* messages received from its child agents, and then projecting out its own variable. In our example problem, agent a_3 computes the optimal utility for each value combination of variables x_1 and x_2 (Figure 5.1(d) top), and sends the utilities to its parent agent a_2 in a *UTIL* message. When the root agent a_1 receives the *UTIL* message from each of its children, it computes the maximum utility of the entire problem.

Observe that the *UTIL propagation* phase of DPOP emulates the BE process in a distributed context [13]. In particular, given a pseudo-tree and its *preorder listing* o, the *UTIL* message generated by each DPOP agent a_i is equivalent to the aggregated and projected function $\hat{f}_i$ in BE when x_i is processed according to the ordering o.

The complexity of DPOP is dominated by the *UTIL propagation* phase, which is exponential in the size of the largest separator set $sep(a_i)$ for all $a_i \in \mathbf{A}$. The other two phases require a polynomial number of linear size messages, and the complexity of the local operations is at most linear in the size of the domain.

5.2.2 GPU-DBE

Our *GPU-based (Distributed) Bucket Elimination* framework, extends BE (respectively DPOP) by exploiting GPGPU parallelism within the *aggregation* and *projection* operations. These operations are responsible for the creation of the functions $\hat{f}_i$ in BE (line 3 of Algorithm 1) and the *UTIL* tables in DPOP (*UTIL* propagation phase), and they dominate the complexity of the algorithms. Thus, we focus on the details of the design and the implementation relevant to such operations. Due to the equivalence of BE and DPOP, we will refer to the *UTIL tables* and to the aggregated and projected functions $\hat{f}$ of Algorithm 2, as well as variables and agents, interchangeably. Notice that the computation of the utility for each value combination in a *UTIL* table is independent of the computation in the other combinations. The use of a GPGPU architecture allows us to exploit such independence, by concurrently exploring several combinations of the *UTIL* table, computed by the aggregation operator, as well as concurrently projecting out variables.

Algorithm 2 illustrates the pseudocode, where we use the following notations: Line numbers in parenthesis denote those instructions required exclusively in the distributed case. Starred line numbers denote those instructions executed concurrently by both the CPU and the GPGPU. The symbols $\leftarrow$ and $\Leftarrow$ denote sequential and parallel (multiple GPU-threads) operations, respectively. If a parallel operation requires

```
Algorithm 2: GPU-(D)BE
(1) Generate pseudo-tree
2   GPU-INITIALIZE( ) ;
3   if $C_{a_i} = \emptyset$ then
4       $UTIL_{x_i} \leftleftarrows$ PARALLELCALCUTILS( ) ;
(5)     Send $UTIL$ message $(x_i, UTIL_{x_i})$ to $P_{a_i}$ ;
6   else
7       Activate UTILMessageHandler(·) ;
(8) Activate VALUEMessageHandler(·) ;
```

```
Procedure UTILMessageHandler($a_k$, $UTIL_{a_k}$)
(9)  Store $UTIL_{a_k}$
10   if received $UTIL$ message from each child $a_c \in C_{a_i}$ then
11       $UTIL_{a_i} \leftleftarrows$ PARALLELCALCUTILS( ) ;
12       if $P_{a_i} = NULL$ then
13           $d_i^* \leftarrow$ CHOOSEBESTVALUE($\emptyset$);
(14)         foreach $a_c \in C_{a_i}$ do
(15)             $VALUE_{a_i} \leftarrow (x_i, d_i^*)$ ;
(16)             Send $VALUE$ message $(a_i, VALUE_{a_i})$ to $a_c$ ;
17       else Send $UTIL$ message $(a_i, UTIL_{a_i})$ to $P_{a_i}$
```

a copy from host (device) to device (host), we write $\overset{D \leftarrow H}{\leftleftarrows}$ ($\overset{H \leftarrow D}{\leftleftarrows}$). Host to device (respectively device to host) memory transfers are performed immediately before (respectively after) the execution of the GPU kernel. Algorithm 2 shows the pseudocode of GPU-(D)BE for an agent a_i. Like DPOP, also GPU-(D)BE is composed of three phases; the first and third phase are executed exclusively in the distributed version. The first phase is identical to that of DPOP (line 1). In the second phase:

- Each agent a_i calls GPU-INITIALIZE() to set up the GPGPU kernel. For example, it determines the amount of global memory to be assigned to each *UTIL* table and initializes the data structures on the GPGPU device memory (line 2). We will discuss it in details in the next section. The GPGPU kernel settings are decided according to the shared memory requirements and the number of registers used by the successive function call, so to maximize the number of blocks that can run in parallel.
- Each agent a_i aggregates the utilities for the functions between its variables and its separator, projects its variable out (line 4), and sends them to its parent (line 5). The *MessageHandlers* of lines 7 and 8 are activated for each new incoming message. The agent repeats this process each time it receives a *UTIL* message from a child (lines 9-16).

By the end of the second phase (line 11), the root agent knows the overall utility for each values of its variable x_i. It chooses the value that results in the maximum utility (line 13). Then, in the distributed version, it starts the third phase by sending to each child agent a_c the value of its variable x_i (lines 14-16). These operations are repeated by every agent receiving a *VALUE* message (lines 18-22). This last phase

```
Procedure VALUEMessageHandler($a_k$, $VALUE_{a_k}$)
(18) $VALUE_{a_i} \leftarrow VALUE_{a_k}$
(19) $d_i^* \leftarrow$ CHOOSEBESTVALUE($VALUE_{a_i}$) ;
(20) foreach $a_c \in C_{a_i}$ do
(21)     $VALUE_{a_i} \leftarrow \{(x_i, d_i^*)\} \cup \{(x_k, d_k^*) \in VALUE_{a_k} \mid x_k \in sep(a_c)\}$ ;
(22)     Send VALUE message $(a_i, VALUE_{a_i})$ to $a_c$ ;
```

is not required in the centralized version, as the value assignment for each variable can be accessed by the root agent directly.

GPGPU Data Structures

In order to fully utilize on the parallel computational power of GPGPUs, the data structures need to be designed in such a way to limit the amount of information exchanged between the CPU host and the GPGPU device, and in order to minimize the accesses to the (slow) device global memory (and ensure that they are coalesced). To do so, each agent identifies the set of relevant *static entities*, i.e., information required during the GPGPU computation, which does not mutate during the resolution process. The static entities are communicated to the GPGPU once at the beginning of the computation. This allows each agent running on a GPGPU device to communicate with the CPU host exclusively to exchange the results of the aggregation and projection processes. The complete set of utility functions, the constraint graph, and the agents ordering, all fall in such category. Thus, each agent a_i stores:

- The set of utility functions involving exclusively x_i and a variable in a_i's separator set: $S_i = \{f_j \in \mathbf{C} \mid x_i \in \mathbf{x}^j \wedge sep(a_i) \cap \mathbf{x}^j \neq \emptyset\}$. For a given function $f_j \in S_i$, its utility values are stored in an array named $gFunc_j$.
- The domain D_i of its variable (for simplicity assumed to be all of equal cardinality).
- The set C_{a_i} of a_i's children.
- The separator sets $sep(a_i)$, and $sep(a_c)$, for each $a_c \in C_{a_i}$.

The GPU-INITIALIZE() procedure of line 2, invoked after the pseudo-tree construction, stores the data structures above for each agent on the GPGPU device. As a technical detail, all the data stored on the GPGPU global memory is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*. In particular, the identifier and scope of the functions in S_i as well as identifiers and separator sets of child agents in C_{a_i} are stored within a single mono-dimensional array. The utility values stored in the rows of each function are padded to ensures that a row is aligned to a memory word—thus minimizing the number of memory accesses.

GPU-INITIALIZE() is also responsible for reserving a portion of the GPGPU global memory to store the values for the agent's *UTIL* table, denoted by $gUtils_i$, and those of its children, denoted by $gChUtils_c$, for each $a_c \in C_{a_i}$. As a technical note, an agent's *UTIL* table is mapped onto the GPGPU device to store only the utility values, not the associated variables values. Its j-th entry is associated with the j-th permutation of the variable values in $sep(a_i)$, in lexicographic order. This strategy allows us to employ a simple perfect hashing to efficiently associate row numbers with variables' values and vice versa. Note that the agent's *UTIL* table size grows exponentially with the size of its separator set; more precisely, after

```
Procedure ParallelCalcUtils( )
23   if $project\_on\_device$ then
24       $gChUTIL_{a_c} \overset{D \leftarrow H}{\Leftarrow} UTIL_{a_c}$   for all $a_c \in C_{a_i}$;
25   $R \leftarrow 0$ ; $UTIL_{a_i} \leftarrow \emptyset$ ;
26   while $R < |D_i|^{sep(a_i)}$ do
27       if $project\_on\_device$ then
28*          $UTIL'_{a_i} \overset{H \leftarrow D}{\Leftarrow}$ GPU-AGGREGATE-PROJECT($R$);
29       else
30*          $UTIL'_{a_i} \overset{H \leftarrow D}{\Leftarrow}$ GPU-AGGREGATE($R$);
31*          $UTIL'_{a_i} \leftarrow$ AGGREGATECH-PROJECT($a_i, UTIL'_{a_i}, UTIL_{a_c}$)   for all $a_c \in C_{a_i}$ ;
32*      $UTIL_{a_i} \leftarrow UTIL_{a_i} \cup$ COMPRESS($UTIL'_{a_i}$);
33       $R \leftarrow R + |UTIL'_{a_i}|$ ;
34   return $UTIL_{a_i}$
```

projecting out x_i, it has $|D_i|^{sep(a_i)}$ entries. However, the GPGPU global memory is typically limited to a few GB (e.g., in our experiments it is 2GB). Thus, each agent, after allocating its static entities, checks if it has enough space to allocate its children's *UTIL* tables and a consistent portion (see next subsection for details) of its own *UTIL* table. In this case, it sets the *project_on_device* flag to true, which signals that both aggregate and project operations can be done on the GPGPU device.[1] Otherwise it sets the flag to false and bounds the device *UTIL* size table to the maximum storable space on the device. In this case, the aggregation operations are performed only partially on the GPGPU device.

Parallel Aggregate and Project Operations

The PARALLELCALCUTILS procedure (executed in lines 4 and 11) is responsible for performing the aggregation and projection operations, harnessing the parallelism provided by the GPGPU. Due to the possible large size of the *UTIL* tables, we need to separate two possible cases and devise specific solutions accordingly:

(a) When the device global memory is sufficiently large to store all a_i's children *UTIL* tables as well as a significant portion of a_i's *UTIL* table[2] (i.e., when $project\_on\_device$ = `true`), both aggregation and projection of the agent's *UTIL* table are performed in parallel on the GPGPU. The procedure first stores the *UTIL* tables received from the children of a_i into their assigned locations in the GPGPU global memory (lines 23-24). It then iterates through successive GPGPU kernel calls (line 28) until the $UTIL_{a_i}$ table is fully computed (lines 26-33). Each iterations computes a certain number of rows of the $UTIL_{a_i}$ table (R serves as counter).

(b) When the device global memory is insufficiently large to store all a_i's children *UTIL* tables as well as a significant portion of a_i's *UTIL* table (i.e., when $project\_on\_device$ = `false`), the agent alternates

[1] If the *UTIL* table of agent a_i does not fit in the global memory, we partition such table in smaller chunks, and iteratively execute the GPGPU kernel until all rows of the table are processed.

[2] In our experiments, we require that at least 1/10 of the *UTIL* table can be stored in the GPGPU. We experimentally observed that a partitioning of the table in at most 10 chunks provides a good time balance between memory transfers and actual computation.

the use of the GPGPU and the CPU to compute *UTIL*$_{a_i}$. The GPGPU is in charge of aggregating the functions in S_i (line 30), while the CPU aggregates the children *UTIL* table,[3] projecting out x_i. Note that, in this case, the *UTIL*$_{a_i}$ storage must include all combinations of values for the variables in $sep(x_i) \cup \{x_i\}$, thus the projection operation is performed on the CPU host. As in the previous case, the *UTIL*$_{a_i}$ is computed incrementally, given the amount of available GPGPU global memory.

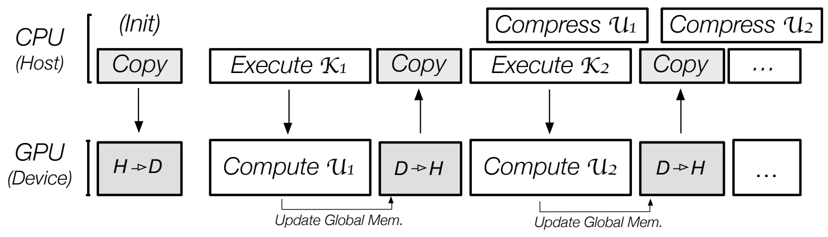


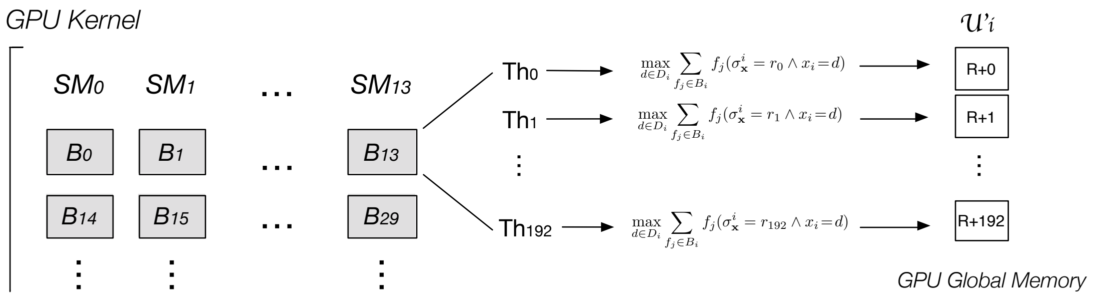
Figure 5.2: Concurrent computation between host and device.

To fully utilize on the use of the GPGPU, we exploit an additional level of parallelism, achieved by running GPGPU kernels and CPU computations concurrently; this is possible when the *UTIL*$_{a_i}$ table is computed in multiple chunks. Figure 5.2 illustrates the concurrent computations between the CPU and GPGPU. After transferring the children *UTIL* tables into the device memory (Init)—in case **(a)** only—the execution of kernel K_1 produces the update of the first chunk of *UTIL*$_{a_i}$, denoted by U_1 in Figure 5.2, which is transferred to the CPU host. The successive parallel operations are performed asynchronously with respect to the GPGPU, that is, the execution of the j-th CUDA kernel K_j ($j > 1$), returns the control immediately to the CPU, which concurrently operates a compression operation on the previously computed *UTIL*$'_{a_i}$ chunk (line 32), referred to as U_{k-1} in Figure 5.2. For case **(b)**, the CPU also executes concurrently the AGGREGATECH-PROJECT of line 31. We highlight the concurrent operations by marking with a * symbol their respective lines in the procedure PARALLELCALCUTILS.

Technical Details: We now describe in more detail how we divide the workload among parallel blocks, i.e., the mapping between the *UTIL* table rows and the CUDA blocks. A total of $T = 64 \cdot k$ ($1 \leq k \leq 16$) threads (a block) are associated to the computation of T permutations of values for $sep(a_i)$. The value k depends on the architecture and it is chosen to maximize the number of concurrent threads running at the same time. In our experiments, we set $k = 3$. The number of blocks is chosen so that the corresponding aggregate number of threads does not exceed the total number of *UTIL*$'_{a_i}$ permutations currently stored in the device. Let h be the number of stream multiprocessors of the GPGPU. Then, the maximum number of *UTIL* permutations that can be computed concurrently is $M = h \cdot T$. In our experiments $h = 14$, and thus, $M = 2688$. Figure 5.6 provides an illustration of the *UTIL* permutations computed in parallel on GPGPU. The blocks B_i in each row are executed in parallel on different SMs. Within each block, a total of (at most) 192 threads operate on as many entries of the *UTIL* table. Such number is bounded by the maximum number of warps that can run in parallel, which in turn is dependent on the characteristic of the hardware and of the kernel (e.g., the number of registers and the amount shared memory required by the kernel play a key role).

The GPGPU kernel procedure is shown in lines 35-49. We surround line numbers with $|\cdot|$ to denote

[3]The CPU aggregates only those child *UTIL* table that could not fit in the GPGPU memory. Those that fit in memory are integrated through the GPGPU computation as done in the previous point.



GPU Kernel
SM0
SM1
SM13
B0
B1
B13
B14
B15
B29
Th0
Th1
Th192
$\max_{d\in D_i} \sum_{f_j\in B_i} f_j(\sigma^i_{\mathbf{x}} = r_0 \wedge x_i = d)$
$\max_{d\in D_i} \sum_{f_j\in B_i} f_j(\sigma^i_{\mathbf{x}} = r_1 \wedge x_i = d)$
$\max_{d\in D_i} \sum_{f_j\in B_i} f_j(\sigma^i_{\mathbf{x}} = r_{192} \wedge x_i = d)$
$\mathcal{U}'_i$
R+0
R+1
R+192
GPU Global Memory

Figure 5.3: GPU kernel parallel computations.

Procedure GPU-Aggregate-Project(R)

```
|35| $r_{id} \leftarrow$ the thread's entry index of $UTIL'_i$;
|36| $d_{id} \leftarrow$ the thread's value index of $D_i$ ;
|37| $\langle|\theta, S_i|, C_{a_i}, sep(x_c)\rangle \leftarrow$ ASSIGNSHAREDMEM()   for all $x_c \in C_{a_i}$
|38| $\theta \leftarrow$ DECODE$(R + r_{id})$ ;
|39| $util \leftarrow -\infty$ ;
40   foreach $d_{id} \in D_i$ do
|41|     $util_{d_{id}} \leftarrow 0$;
|42|     foreach $f_j \in S_i$ do
|43|         $\rho_j \leftarrow$ ENCODE$(\theta_{\mathbf{x}^j} \mid x_i = d_{id})$ ;
|44|         $util_{d_{id}} \leftarrow util_{d_{id}} + gFunc_j[\rho_j]$ ;
45       foreach $a_c \in C_{a_i}$ do
46           $\rho_c \leftarrow$ ENCODE$(\theta_{sep(a_c)} \mid x_i = d_{id})$ ;
47           $util_{d_{id}} \leftarrow util_{d_{id}} + gChUtils_c[\rho_c]$ ;
|48|     $util \leftarrow \max(util, util_{d_{id}})$ ;
|49| $gUtils_i[r_{id}] \leftarrow util$;
```

parts of the procedure executed by case **(b)**. The kernel takes as input the number R of the *UTIL* table permutations computed during the previous kernel calls. Each thread identifies its entry index r_{id} within the table chunk $UTIL'_{a_i}$ (line 35). It then assigns the shared memory allocated to local arrays to store the static entities S_i, C_{a_i}, and $sep(a_c)$, for each $a_c \in C_{a_i}$. In addition it reserves the space θ to store the assignments corresponding to the *UTIL* permutation being computed by each thread, which is retrieved using the thread entry index and the offset R (line 38). DECODE implements a *minimal perfect hash function* to convert the entry index of the *UTIL* table to its associated variables value permutation. Each thread aggregates the functions in S_i (lines 42-44) and the *UTIL* tables of a_i's children (lines 45-47), for each element of its domain (lines 40-48). The ENCODE routine converts a given assignments for the variables in the scope of a function f_j (line 43), or in the separator set of child a_c (line 46), to the corresponding array index, sorted in lexicographic order. The value for the variable x_i within each input, is updated at each iteration of the for loop. The projection operation is executed in line 48. Finally, the thread stores the best utility in the corresponding position of the array $gUtils_i$

The GPU-AGGREGATE procedure (called in line 30), is illustrated in lines 35-49—line numbers surrounded by $|\cdot|$. Each thread is in charge of a value combination in $sep(a_i) \cup \{x_i\}$, thus, the **foreach** loop of lines 40-48 is operated in parallel by $|D_i|$ threads. Lines 45-47 are not executed. The AGGREGATECH-PROJECT procedure (line 31), which operates on the CPU, is similar to the GPU-AGGREGATE-PROJECT procedure, except that lines 36-37, and 42-44, are not executed.

The proposed kernel has been the result of several investigations. We experimented with other levels of parallelism, e.g., by unrolling the for-loops among groups of threads. However, these modifications create divergent branches, which degrade the parallel performance. We experimentally observed that such degradation worsen consistently as the size of the domain increases.

5.2.3 Theoretical Analysis

We now report some observation on the complexity, completeness, and correctness of our GPU-DBE, which directly follow from the complexity, completeness and correctness of BE and DPOP.

Corollary 5. *GPU-DBE requires the same number of messages as those required by DPOP, and it requires messages of the same size as those required by DPOP.*

Corollary 6. *The UTIL messages constructed by each GPU-DBE agent are identical to those constructed by each corresponding DPOP agent.*

The above observations follow from the pseudo-tree construction and VALUE propagation GPU-DBE phases, which are identical to those of DPOP. Thus, their corresponding messages and message sizes are identical in both algorithms. Moreover, given a pseudo-tree, each DPOP/GPU-DBE agent computes the *UTIL* table containing each combination of values for the variables in its separator set. Thus, the *UTIL* messages of GPU-DBE and DPOP are identical.

Corollary 7. *The memory requirements of GPU-(D)BE is, in the worst case, exponential in the induced width of the problem (for each agent).*

This observation follows from the equivalence of the *UTIL* propagation phase of DPOP and BE [13] and from Corollary 6.

Corollary 8. *GPU-(D)BE is complete and correct.*

The completeness and correctness of GPU-(D)BE follow from the completeness and correctness of BE [25] and DPOP [96].

5.2.4 Related Work

The use of GPGPUs to solve difficult combinatorial problems has been explored by several proposals in different areas of constraint solving and optimization [21]. For instance, Meyer *et al.* [65] proposed a multi-GPGPU implementation of the *simplex tableau* algorithm which relies on a vertical problem decomposition to reduce communication between GPGPUs. In constraint programming, Arbelaez and Codognet [3] proposed a GPU-based version of the *Adaptive Search* that explores several *large neighborhoods* in parallel, resulting in a speedup factor of 17. Campeotto *et al.* [16] proposed a GPU-based framework that exploits both parallel propagation and parallel exploration of several large neighborhoods using local search techniques, leading to a speedup factor of up to 38. The combination of GPGPUs with dynamic programming has also been explored to solve different combinatorial optimization problems. For instance, Boyer *et al.* [12] proposed the use of GPGPUs to compute the classical DP recursion step for the knapsack problem, which led to a speedup factor of 26. Pawłowski *et al.* [90] presented a DP-based solution for the *coalition structure formation problem* on GPGPUs, reporting up to two orders of magnitude of speedup. Differently from other proposals, our approach aims at using GPGPUs to exploit SIMT-style parallelism from DP-based methods to solve general COPs and DCOPs.

5.2.5 Experimental Evaluation

We compare our centralized and distributed versions of GPU-(D)BE with BE [25] and DPOP [96] on binary constraint networks with *random*, *scale-free*, and *regular grid* topologies. The instances for each topology are generated as follows:

- **Random:** We create an n-node network, whose density p_1 produces $\lfloor n\,(n-1)\,p_1 \rfloor$ edges in total. We do not bound the tree-width, which is based on the underlying graph.
- **Scale-free:** We create an n-node network based on the Barabasi-Albert model [5]: Starting from a connected 2-node network, we repeatedly add a new node, randomly connecting it to two existing nodes. In turn, these two nodes are selected with probabilities that are proportional to the numbers of their connected edges. The total number of edges is $2\,(n-2)+1$.
- **Regular grid:** We create an n-node network arranged as a rectangular grid, where each internal node is connected to four neighboring nodes, while nodes on the grid edges (respectively corners) are connected to two (respectively three) neighboring nodes.

We generate 30 instances for each topology, ensuring that the underlying graph is connected. The utility functions are generated using random integer costs in $[0, 100]$, and the constraint tightness (i.e., ratio of entries in the utility table different from $-\infty$) p_2 is set to 0.5 for all experiments. We set as default parameters, $|\mathbf{A}| = |\mathbf{X}| = 10$, $|D_i| = 5$ for all variables, and $p_1 = 0.3$ for random networks, and $|\mathbf{A}| = |\mathbf{X}| = 9$ for regular grids. Experiments for GPU-DBE are conducted using a multi-agent DCOP simulator, that simulates the concurrent activities of multiple agents, whose actions are activated upon receipt of a message. We use the publicly-available implementation of DPOP available in the FRODO framework v.2.11 [69], and we use the same framework to run the BE algorithm, in a centralized setting.

Since all algorithms are complete, our focus is on runtime. Performance of the centralized algorithms are evaluated using the algorithm's wallclock runtime, while distributed algorithms' performances are evaluated using the *simulated runtime* metric [111]. We imposed a timeout of 300s of wallclock (or simulated) time and a memory limit of 32GB. Results are averaged over all instances and are statistically significant with p-values $< 1.638\,e^{-12}$.[4] These experiment are performed on an *AMD Opteron 6276*, 2.3GHz, 128GB of RAM, which is equipped with a GPGPU device *GeForce GTX TITAN* with 14 multiprocessors, 2688 cores, and a clock rate of 837MHz.

[4] t-test performed with null hypothesis: GPU-based algorithms are faster than non-GPU ones.

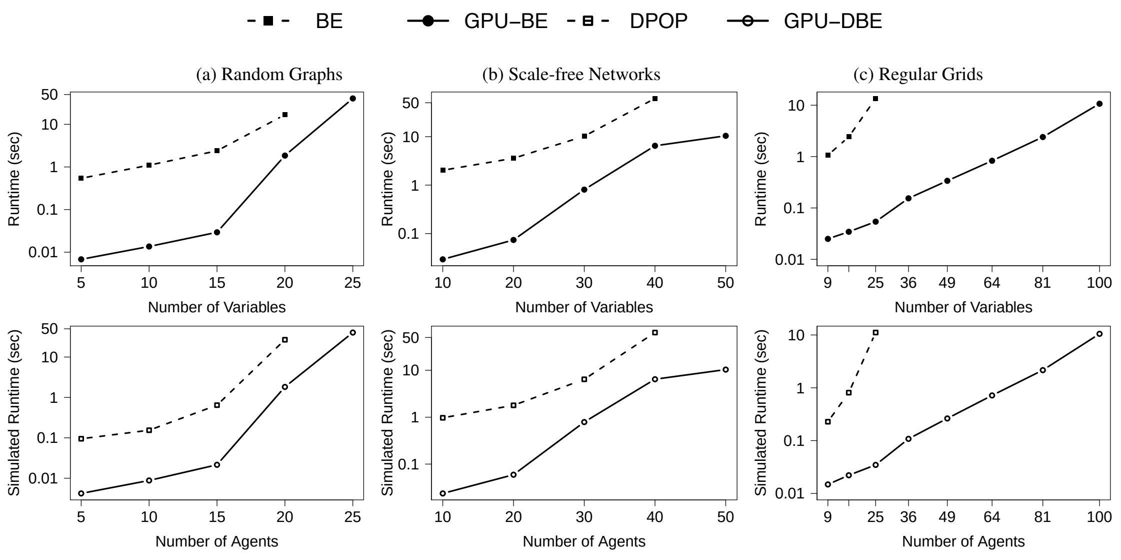


Figure 5.4: Runtimes for COPs (top) and DCOPs (bottom) at varying number of variables/agents.

Figure 5.4 illustrates the runtime, in seconds, for random (a), scale-free (b), and regular grid (c) topologies, varying the number of variables (respectively agents) for the centralized (respectively distributed) algorithms. The centralized algorithms (BE and GPU-BE) are shown at the top of the figure, while the distributed algorithms (DPOP and GPU-DBE) are illustrated at the bottom. All plots are in log-scale. We make the following observations:

- The GPU-based DP-algorithms (for both centralized and distributed cases) are consistently faster than the non-GPU-based ones. The speedups obtained by GPU-BE vs. BE are, on average, and minimum (showed in parenthesis) 69.3 (16.1), 34.9 (9.5), and 125.1 (42.6), for random, scale-free, and regular grid topologies, respectively. For the distributed algorithms, the speedups obtained by GPU-DBE vs. DPOP are on average (minimum) 44.7 (14.7), 22.3 (8.2), and 124.2 (38.8), for random, scale-free, and regular grid topologies, respectively.
- In terms of scalability, the GPU-based algorithms scale better than the non-GPU-based ones. In addition, their scalability increases with the level of structure exposed by each particular topology. On random graphs, which have virtually no structure, the GPU-based algorithms reach a timeout for instances with small number of variables (25 variables—compared to 20 variables for the non-GPU-based algorithms). On scale-free networks, the GPU-(D)BE algorithms can solve instances up to 50 variables,[5] while BE and DPOP reach a timeout for instances greater than 40 variables. On regular grids, the GPU-based algorithms can solve instances up to 100 variables, while the non-GPU-based ones, fail to solve any instance with 36 or more variables.
  We relate these observations to the size of the separator sets and, thus, the size of the *UTIL* tables that are constructed in each problem. In our experiments, we observe that the average sizes of the separator sets are consistently larger in random graphs, followed by scale-free networks, followed by regular grids.
- Finally, the trends of the centralized algorithms are similar to those of the distributed algorithms: The

[5]With 60 variables, we reported 12/30 instances solved for GPU-(D)BE.

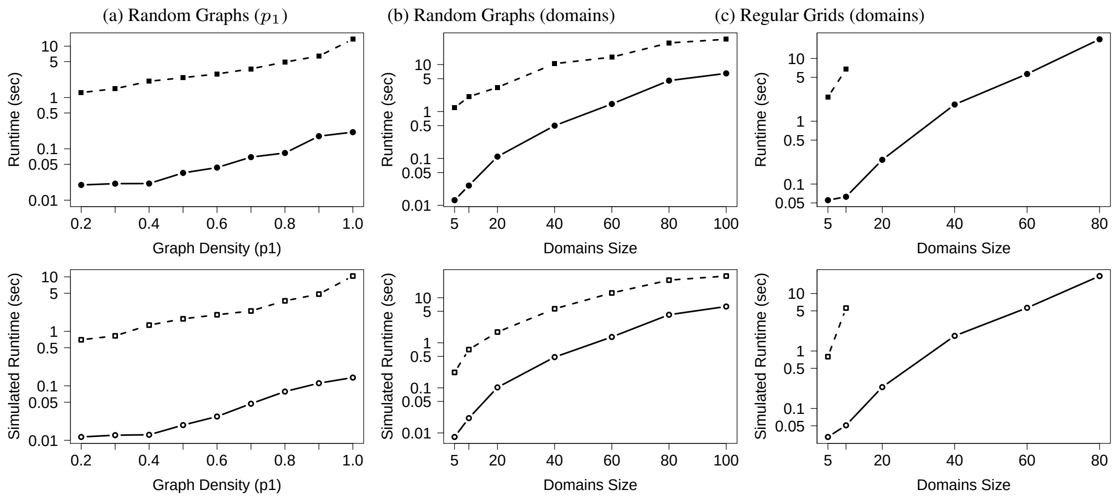


Figure 5.5: Runtimes for COPs (top) and DCOPs (bottom) at varying number of variables/agents.

simulated runtimes of the DCOP algorithms are consistently smaller than the wallclock runtimes of the COP ones.

Figure 5.5 illustrates the behavior of the algorithms when varying the graph density p_1 for the random graphs (a), and the domains size for random graphs (b) and regular grids (c). As for the previous experiments, the centralized (respectively distributed) algorithms are shown on the top (respectively bottom) of the figure. We can observe:

- The trends for the algorithms runtime, when varying both p_1 and domains size, are similar to those observed in the previous experiments.
- GPU-(D)BE achieves better speed-up for smaller p_1 (Figure 5.4 (a)). The result is explained by observing that small p_1 values correspond to smaller induced width of the underlying constraint graph. In turn, for small p_1 values, GPU-(D)BE agents construct smaller *UTIL* tables, which increases the probability of performing the complete inference process on the GPU, through the GPU-AGGREGATE-PROJECT procedure. This observation is also consistent with what observed in the previous experiments in terms of scalability.
- GPU-(D)BE achieves greater speedups in presence of large domains. This is due to the fact that large domains correspond to large *UTIL* tables, enabling the GPU-based algorithms to exploit a greater amount of parallelism, provided that the *UTIL* tables can be stored in the global memory of the GPGPU.

5.3 Accelerating MVA-based algorithm on GPGPUs

In Chapter 3, we introduced the MVA decomposition for DCOPs with multi-variable agents. Such decomposition exploits co-locality of each agent's variables to enable a separation between the agents' *local subproblems* and the DCOP *global problem*. In addition, the MVA decomposition facilitates the use of a hierarchical parallel model, as the whole DCOP can be solved asynchronously, and each individual agent's subproblem can be solved exploiting parallelism. In this section we explore centralized solving sampling-based algorithms, which benefit form the use of GPGPUs, and integrate them within the MVA DCOP framework.

The use of hierarchical parallel solutions is motivated by the observation that the search for the best local solution for each row of the MVA_TABLE is independent of the search for another row and, as such, they can be performed in parallel (see section 3.2). This observation finds a natural fit for SIMT processing and, therefore, in addition to the CPU versions of Gibbs sampling, detailed in section 3.2.3, we provide its GPGPU counterpart, and propose a new framework which can handle general MCMC sampling algorithms, accelerated through the use of GPGPUs. The use of GPGPUs allows us to speed up the local optimization process and, consequently, reduces the overall DCOP solving time.

5.3.1 Notation and Definitions

In this section, we first give some background on Markov Chains and introduce general properties that need to be satisfied by Markov chains to guarantee convergence to a stationary distribution. We then define the objective of Markov Chain Monte Carlo (MCMC) algorithms to our purpose. Finally, we provide a mapping from a *Maximum a Posteriori* (MAP) estimation problem to a *Distributed Constraint Optimization Problem* (DCOP) using general assumptions from a broad class of MCMC algorithms.

Markov Chains

Definition 16 (Markov chain)**.** *A Markov chain is a collection of random variables* $\mathbf{Z} = (\mathbf{z}^0, \mathbf{z}^1, \ldots, \mathbf{z}^t, \ldots)$, *with* $\mathbf{z}^t \in \mathbf{D} \subseteq \mathbb{R}$ *having the property that, given the present, the future is conditionally independent of the past. Formally,*

$$P(z^{t+1} = s \mid z^1 = s_1, z^2 = s_2, \ldots, z^t = s^t) = P(z^{t+1} = s \mid z^t = s_t),$$

if both conditional probabilities are well defined, i.e. if $P(z_1 = s_1, \ldots, z^t = s_t) > 0$.

The possible values of s_i form a countable set $\mathbf{S}$ called the *state space* of the chain.

We now introduce the structural properties that are required for a Markov chain to guarantee convergence to a stationary distribution π.

Let $\mathbf{Z} = (\mathbf{z}^0, \mathbf{z}^1, \ldots, \mathbf{z}^t, \ldots)$, with $\mathbf{z}^t \in \mathbf{D} \subseteq \mathbb{R}$ be a Markov chain with finite state space $\mathbf{S} = \{s_1, s_2, \ldots, s_L\}$ and a $L \times L$ *transition matrix* T whose entries are all non-negative and such that for each state $s_i \in \mathbf{S}$, $\sum_{s_j \in \mathbf{S}} T_{ij} = 1$, which defines the probability of transiting from one state to another as

$$P(\mathbf{z}^{t+1} = s_j \mid \mathbf{z}^t = s_i) = T_{ij}.$$

We denote with T^m the probability of moving from a state $\mathbf{z}^{\mathbf{0}}$ to a state $\mathbf{z}^m$ in m time steps.

Definition 17 (Irreducibility)**.** *A Markov chain is said to be* irreducible *if it is possible to reach any state to any other using only transitions of positive probability. Formally,*

$$\forall s_i, s_j \in \mathbf{S}, \exists m < \infty \,.\, P(\mathbf{z}^{t+m} = s_j \mid \mathbf{z}^t = s_i)$$

for a given instance t.

Definition 18 (Periodicy). *A state $s_i \in \mathbf{S}$ has a period k if any return of the chain in it is possible with multiple of k time steps. The period of a state is defined as*

$$k = gcd\{t : P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0\}$$

where gcd is the greatest common divisor. A state is said to be aperiodic if $k = 1$, that is, visits of the Markov chain to such state can occur at irregular times: $P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0$. A Markov chain is said to be aperiodic *if every state in $\mathbf{S}$ is aperiodic.*

Note that for an irreducible Markov chain, if at least one state is aperiodic, then the whole Markov chain is aperiodic.

Definition 19 (Reaching Time). *The* reaching time *τ_s of a state $s \in \mathbf{S}$ is the first (positive) time at which a chain visits that state. Formally,*

$$\tau_s := \min\{t \geq 1 \mid \mathbf{z}^t = s\}.$$

Lemma 5. *For any states s_i and s_j of an irreducible Markov chain, the expected first return time for a state s_j from a state s_i occurs in a finite amount of steps, that is*

$$\mathbf{E}_{s_j}(\tau_{s_i}) < \infty.$$

Lemma 6. *Given a Markov chain defined in a finite state space $\mathbf{S}$, with transition matrix P, and for a given initial state of the chain $\mathbf{z}^0 = s_0$, if P is irreducible and aperiodic, then*

$$\exists t < \infty, \forall m \geq t : s_0 T^m = \pi$$

and π is unique. Moreover, for all $s \in \mathbf{S}$, $\pi(s) > 0$ and

$$\pi(s) = \frac{1}{\mathbf{E}_{\mathbf{s}}(\tau_s)}.$$

The above lemma expresses that given enough time, the chain converges to a unique stationary distribution π.

Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) sampling algorithms are commonly used to solve the *Maximum a Posteriori* (MAP) probability estimation problem—a mode of the posterior distribution—once the probability distribution has converged to its stationary point. Suppose we have a joint probability distribution $\pi(\mathbf{z})$ over n variables, with $\mathbf{z} = z_1, \ldots, z_n$, and $z_i \in \mathbb{R}$, which we are interested to approximate. Sampling algorithms are often used to examine posterior distributions as they provide ways of generating samples with the property that the empirical distribution of the samples approximate the posterior distribution π. It is not often the case that one can sample directly from the posterior distribution obtaining an independent and identically distributed (i.i.d.) sample from π. When sampling directly from the posterior distribution is difficult, due to the high dimensionality or because computing the posterior may be computationally intense, one can use a proposal distribution q which approximates the posterior π up to some normalizing constant, and performs a dependent sample, such as the sample path of a Markov chain. MCMC algorithms generate a sample path from a Markov chain that has π as its stationary distribution.

Algorithm 2: METROPOLIS-HASTING($\mathbf{z}$)

```
50 $\mathbf{z}^{(0)} \leftarrow$ INITIALIZE($\mathbf{z}$)
51 for $t = 1$ to $T$ do
52     $\mathbf{z}^* \leftarrow$ SAMPLE($q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$)
53     $\mathbf{z}^{(t)} \leftarrow \begin{cases} \mathbf{z}^* & \text{with } p = \min(1, \frac{\tilde{\pi}(\mathbf{z}^*)q(\mathbf{z}^{(t-1)}, \mathbf{z}^*)}{\tilde{\pi}(\mathbf{z}^{(t-1)})q(\mathbf{z}^*, \mathbf{z}^{(t-1)})}) \\ \mathbf{z}^{(t-1)} & \text{with } 1-p \end{cases}$
```

```
54 for $i = 1$ to $n$ do
55     $z_i^t \leftarrow$ SAMPLE($\frac{1}{Z_\pi}\tilde{\pi}(z_i \mid z_1^t, \dots, z_{i-1}^t, z_{i+1}^{t-1}, \dots, z_n^{t-1})$)
```

In more details, suppose that it is easy to evaluate $\pi(\mathbf{z})$ for any given $\mathbf{z}$ up to some normalizing constant Z_π, such that: $\pi(\mathbf{z}) = \frac{1}{Z_\pi}\tilde{\pi}(\mathbf{z})$, where $\tilde{\pi}(\mathbf{z})$ can be easily computed but Z_π may be unknown or hard to evaluate. In order to draw the samples $\mathbf{z}$ to be fed to $\tilde{\pi}(\cdot)$, we use a *proposal distribution* $q(\mathbf{z}|\mathbf{z}^{(\tau)})$, from which we can easily generate samples, each depending on the current state $\mathbf{z}^{(\tau)}$ of the process. The latter can be interpreted as saying that when the process is in the state $\mathbf{z}^{(\tau)}$, we can generate a new state $\mathbf{z}$ from $q(\mathbf{z} \mid \mathbf{z}^{(\tau)})$. The proposal distribution is thus used to generate a sequence of samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \dots$, which forms a Markov chain.

MCMC Methods

Let us describe two popular MCMC algorithm—Gibbs [38] and Metropolis-Hastings [53, 77].

Algorithm 2 shows the pseudocode of the *Metropolis-Hastings* algorithm. It first initializes $\mathbf{z}^{(0)}$ to any arbitrary value of the variables $z_1, \dots, z_n$ (line 1). Then, it iteratively generates a candidate $\mathbf{z}^*$ for $\mathbf{z}^{(t)}$ by sampling from the proposal distribution $q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$ (line 3). The candidate sample is then accepted with probability p defined in line 4. If the candidate sample is accepted, then $\mathbf{z}^{(t)} = \mathbf{z}^*$, otherwise $\mathbf{z}^{(t-1)}$ is left unchanged. This process continues for a fixed number of iterations or until convergence [103] is achieved.

The *Gibbs* sampling algorithm is a special case of the Metropolis-Hastings algorithm, where line 3 is replaced by lines 5-6. Additionally, note that Gibbs requires the computation of the normalizing constant Z_π while Metropolis-Hasting does not, as the calculation of the proposal distribution does not require that information. This is desirable when the computation of the normalizing constant becomes prohibitive (e.g., with increasing problem dimensionality).

Maximum A-Posteriori to DCOP Mapping

Recently, Nguyen et al. [84] has shown that DCOPs can be mapped to MAP estimation problems. Thus, MCMC algorithms can be used to solve DCOPs as well. We now show how to extend this mapping to the general case of multivariable DCOP functions.

Consider a MAP problem on a *Markov Random Field* (MRF). An MRF is a set of random variables having the *Markov property*—the conditional probability distribution of future states of the process do not depends on other states other than the current one—and it can be described by an undirected graph (V, E). Formally an MRF is defined by

- a set of random variables $\mathbf{z} = \{z_i \mid \forall i \in V\}$, where each random variable z_i is defined over a finite domain D_i. Each random variable z_i is visualized through a node $i \in V$.
- A set of potential functions $\theta = \{\theta_i(z_k) \mid z_k \in C_i\}$, where C_i refers to a set of nodes of V denoting a clique which includes node i.

Let the joint probability distribution $\pi(z_k = d_k : z_k \in C_i)$ be defined as $\exp[\theta_i(z_k \mid z_k = d_k \in C_i)]$. For ease of presentation we denote as $\pi(z_k : z_k \in C_i)$ the joint probability of the random variables $z_k \in C_i$ and mean $\pi(z_k = d_k : z_k \in C_i)$.

A full-joint distribution of $\mathbf{z}$ has the probability:

$$\pi(\mathbf{z}) = \frac{1}{Z} \prod_{C_i \in \mathbf{C}} \exp\left[\theta_i(z_k : z_k \in C_i)\right] \tag{5.1}$$

$$= \frac{1}{Z} \exp\left[\sum_{C_i \in \mathbf{C}} \theta_i(z_k : z_k \in C_i)\right] \tag{5.2}$$

where $\mathbf{C}$ is the set of all cliques in (V, E) and Z is the normalizing constant for the density. The objective of a MAP estimation problem is to find the mode of $\pi(\mathbf{z})$, which is equivalent to find a complete assignment $\mathbf{z}$ that maximizes the function:

$$F(\mathbf{z}) = \sum_{C_i \in \mathbf{C}} \theta_i(z_k : z_k \in C_i)$$

which is also objective of a DCOP, where each potential function θ_i correspond to a utility function f_i and the associated clique C_i to the scope of the function f_i.

Therefore, if T is an MCMC sampling method that constructs a Markov chain with stationary distribution π to solve the associated MAP estimation problem, then, we can use the complete solution $\mathbf{z}$ returned to solve the corresponding DCOP.

Notice that sufficient conditions for T to converge to π are irreducibility and aperiodicity. The *Gibbs* and *Metropolis-Hastings* sampling algorithms exhibit extremely weak sufficient conditions to guarantee convergence [103]. Namely, the Gibbs proposal distribution needs to ensure lower semi-contiguity at 0 and be locally bounded, while for the Metropolis Hasting, it is sufficient that the domain of the definition of the proposal distribution q coincide with that of π.

5.3.2 Distributed Markov Chain Monte Carlo Sampling MVA Framework

We now describe our *Distributed MCMC* (DMCMC) framework, which extends centralized MCMC sampling algorithms and DPOP. At a high level, its operations are similar to the operations of DPOP except that the computation of the utility tables sent by agents during the UTIL phase is done by sampling with GPGPUs. Notice that the computation of each row in a utility table is independent of the computation in the other rows. Thus, DMCMC exploits this independence and samples the utility in each row in parallel.

Algorithm 2 shows the pseudocode of DMCMC for an agent a_i. It takes as inputs R, the number of sampling runs to perform from different initial value assignments, and T, the number of sampling trials.

Algorithm 2: DMCMC(R, T)

```
56 Generate pseudo-tree
57 GPU-INITIALIZE( )
58 $\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle \leftarrow$ GPU-MCMC-SAMPLE($R, T$)
59 $UTIL_{a_i} \leftarrow$ GET-BEST-SAMPLE($\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle$)
60 if $C_{a_i} = \emptyset$ then
61     $UTIL_{a_i} \leftarrow$ CALCUTILS( )
62     Send $UTIL$ message $(a_i, UTIL_{a_i})$ to $P_i$
63 Activate UTILMessageHandler($\cdot$)
64 Activate VALUEMessageHandler($\cdot$)
```

Procedure VALUEMessageHandler(a_k, $VALUE_{a_k}$)

```
65 $VALUE_{a_i} \leftarrow VALUE_{a_k}$
66 for $x_i^j \in L_i$ do $d_i^{j*} \leftarrow$ CHOOSEBESTVALUE($VALUE_{a_i}$) for $a_c \in C_{a_i}$ do
67     $VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\} \cup \{(x_k, d_k^*) \in VALUE_{a_k} \mid x_k \in sep(a_c)\}$
68     Send $VALUE$ message $(a_i, VALUE_{a_i})$ to $a_c$
```

Like DPOP, DMCMC also exhibits three phases. The first phase is identical to that of DPOP (line 56). In the second phase:

- Each agent a_i calls GPU-INITIALIZE() to set up the GPGPU kernel specifics (e.g., number of threads and amount shared memory to be assigned to each block, and to initialize the data structures on the GPGPU device memory) (line 57). The GPGPU kernel settings are decided according to the shared memory requirements and the number of registers used by the successive function call, so to maximize the number of blocks that can run in parallel.
- Each agent a_i, *in parallel*, calls GPU-MCMC-SAMPLE() which performs the local MCMC sampling process to compute the best utility and the corresponding solution (value assignments for all non-boundary local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the boundary variables $x_i^k \in B_i$ (line 58). This computation process is done via sampling with GPGPUs and the results are

Procedure UTILMessageHandler(a_k, $UTIL_{a_k}$)

```
69 Store $UTIL_{a_k}$
70 if received $UTIL$ message from each child $a_c \in C_{a_i}$ then
71     $UTIL_{a_i} \leftarrow$ CALCUTILS( )
72     if $P_{a_i} = NULL$ then
73         for $x_i^j \in L_i$ do $d_i^{j*} \leftarrow$ CHOOSEBESTVALUE($\emptyset$) for $a_c \in C_{a_i}$ do
74             $VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\}$
75             Send $VALUE$ message $(a_i, VALUE_{a_i})$ to $a_c$
76     else Send $UTIL$ message $(a_i, UTIL_{a_i})$ to $P_{a_i}$
```

```
**Function** CalcUtils( )
77 $UTIL_{sep} \leftarrow$ utilities for all value combinations of $x_i \in B_i \cup sep(a_i)$
78 $UTIL_{a_i} \leftarrow \textsc{Join}(UTIL_{a_i}, UTIL_{sep}, UTIL_{a_c})$ for all $a_c \in C_{a_i}$
79 $UTIL_{a_i} \leftarrow \textsc{Project}(a_i, UTIL_{a_i})$
80 **return** $UTIL_{a_i}$
```

```
**Procedure** GPU-MCMC-Sample($R, T$)
81 $\langle \mathbf{z}, \mathbf{z}^*, [q, Z_\pi], G_i \rangle \leftarrow \textsc{AssignSharedMem}()$
82 $r_{id} \leftarrow$ the thread's row index of $M_i$
83 $\mathbf{z} \overset{|L_i|}{\leftleftarrows} M_i[r_{id}]$
84 $\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j}) \rangle$
85 **for** $t = 1$ **to** $T$ **do**
86     $\mathbf{z} \overset{k}{\leftleftarrows} \textsc{Sample}(q(\mathbf{z} \mid \mathbf{z}^{(t-1)}))$ w/ prob. $\min\{1, \frac{\tilde{\pi}(\mathbf{z})}{\tilde{\pi}(\mathbf{z}^{(t-1)})}\}$
87     $util \leftarrow \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})$
88     **if** $util > util^*$ **then** $\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, util \rangle$
89 $\langle M_i^R[r_{id}], U_i^R[r_{id}] \rangle \leftarrow \langle \mathbf{z}^*, util^* \rangle$
```

then transferred from the device to the host (line 10). In our example in Figure 3.1, agent a_3 computes that its best utility is 20 if its boundary variable $x_6 = 0$ and 8 if $x_6 = 1$. This utility table is stored in $UTIL_{a_i}$. Note that all the agents call this procedure immediately after the pseudo-tree is constructed. In contrast, agents in DPOP compute the best utility only after receiving UTIL messages from all children agents.

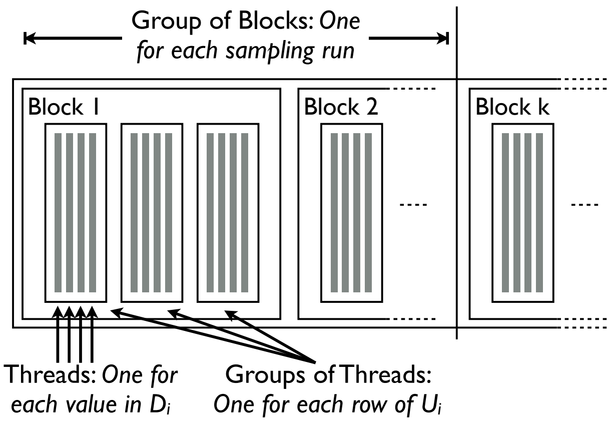
- Each agent a_i computes the utilities for the constraints between its variables and its separator, joins them with the sampled utilities (line 61), and sends them to its parent (line 62). The agent repeats this process each time it receives a UTIL message from a child (lines 20-27).

By the end of the second phase (line 23), like in DPOP, the root agent knows the overall utility for each combination of values of its variables $x_i^j \in B_i$. It chooses its best value combination that results in the maximum utility (line 73), and starts the third phase by sending to each child agent a_c the values of variables $x_i^j \in sep(a_c)$ that are in the separator of the child (lines 73-75). The *MessageHandlers* of lines 63 and 64 are activated for any new incoming message.

GPGPU Data Structures

In order to fully utilize on the parallel computational power of GPGPUs, the data structures need to be designed in such a way to limit the amount of information exchanged between the CPU host and the GPGPU devices. Each DMCMC agent stores all the information it needs in its local variables in the global memory of the GPGPU devices. This allows each agent running on a GPGPU device to communicate with the CPU host only once, which is at the end of the sampling process, to transfer the results. Each agent a_i maintains the following information:

- Its *local* variables $L_i \subseteq \mathbf{X}$.



Group of Blocks: One for each sampling run
Block 1
Block 2
Block k
Threads: One for each value in Di
Groups of Threads: One for each row of Ui

Figure 5.6: Parallelization Illustration

- Its *boundary* variables $B_i \subseteq L_i$.
- The domains of its local variables, D_i (assumed to have all equal size for simplicity).
- The MVA_TABLE M_i of size $|D_i|^{|B_i|} \times |L_i|$, where the j-th row is associated with the j-th permutation of the boundary variable values, in lexicographic order, and the k-th column is associated with the k-th variable in L_i. The MVA_TABLE columns associated with the local variables in L_i are initialized with random value assignments in $[0, D_i - 1]$. At the end of the sampling process it contains the converged domain values of the local variables for each value combination of the boundary variables.
- A vector U_i of size $|D_i|^{|B_i|}$, which stores the utilities of the solutions in M_i.
- The *local constraint graph* G_i, which includes the local variables L_i and constraints between local variables.

The GPU-INITIALIZE() procedure of line 57 stores the data structures above for each agent on its CUDA device. All the data stored on the GPGPU devices is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*. The set of local variables L_i are ordered, for convenience, in lexicographic order and so that the boundary variables B_i are listed first.

Local Sampling Process

The GPU-MCMC-SAMPLE procedure of line 58 is the core of the local sampling algorithm, and can be performed by any MCMC sampling method. It executes T sampling trials for the subset of non-boundary local variables $L_i \setminus B_i$ of agent a_i. Since the MCMC sampling procedure is stochastic, we can run R parallel sampling processes with different initial value assignments and take the best utility and corresponding solution across all runs. Each parallel run is executed by a *group of CUDA blocks*. Independent operations within each sample are also exploited in parallel using *groups of threads* within each block. For example, the proposal distribution adopted by Gibbs is computed using $|D_i|$ parallel *threads*. Figure 5.6 illustrates the different parallelizations performed by the GPU-MCMC-Sample process with Gibbs.

The general GPU-MCMC-Sample procedure is shown in lines 81-89 and we use the symbols $\leftarrow$ and $\overset{k}{\Leftarrow}$ to denote sequential (single thread) and parallel (k threads) operations, respectively. We also denote with n the size of the state $\mathbf{z}$ being sampled, with $n = |L_i| - |B_i|$. The function takes in as inputs the

Procedure CUDA Gibbs Proposal Distribution Calculation

```
90 $d_{id} \leftarrow$ the thread's value index of $D_i$
91 for $k = |B_i|$ to $|L_i| - 1$ do
92     $q[d_{id}] \overset{|D_i|}{\Leftarrow} \exp\left[\sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})\right]$
93     $Z_\pi \leftarrow \sum_{i=0}^{|D_i|-1} q[i]$
94     $q[d_{id}] \overset{|D_i|}{\Leftarrow} q[d_{id}] \cdot \frac{1}{Z_\pi}$
95     $\mathbf{z} \leftarrow$ SAMPLE$(q(\mathbf{z} \mid \mathbf{z}^{(t-1)}))$
```

number of desired sampling trials T and the number of parallel sampling runs R. It first assigns the shared memory allocated to the arrays $\mathbf{z}$ and $\mathbf{z}^*$, which are used to store the current and best sample of value assignments for all local variables, respectively; the local constraint graph G_i; and, if the MCMC sampling algorithm requires computing the normalization constant of the proposal distribution explicitly, the array q and Z_π, which are used to store the probabilities for each value of the non-boundary local variables and the normalization constant, respectively (line 81).

Each thread identifies its row index r_{id} of the MVA_TABLE M_i, initializes its sample with the values stored in $M_i[r_{id}]$, calculates the utility for that sample, and stores the initial sample and utility as the best sample and utility found so far (lines 82-84). It then runs T sampling trials, where in each trial, it samples a new state $\mathbf{z}$ from a proposal distribution $q(\mathbf{z} \,|\, \mathbf{z}^{(t-1)})$ and updates that state according to the accept/reject probabilities described in the MCMC background (line 86).

The proposal distribution q and the accept/reject probabilities depend on the choice of MCMC algorithm. We now describe them for Metropolis-Hasting and Gibbs.

- **Metropolis-Hastings**: The proposal distribution that we adopt is a multivariate normal distribution $q \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\boldsymbol{\mu}$ being a n-dimensional vector of mean values, where each component $\mu_j^{(t)}$ has the value of the corresponding component in the previous sample $z_j^{(t-1)}$ and $\boldsymbol{\Sigma}$ is the covariance matrix defined with the only non-zero elements being their diagonal ones and set to be all equal to $\sqrt{D_i}$. We compute the proposal distribution q using n parallel threads. The proposal distribution for Metropolis-Hastings is symmetric and, thus, the accept/reject probabilities are simplified as shown in line 86.
- **Gibbs**: For Gibbs, line 86 needs to be replaced with lines 41-46. Gibbs sequentially iterates through all the non-boundary local variable $x_k \in L_i \setminus B_i$ and computes in parallel the probability $q[d_{id}]$ of each value d_{id} according to the equation:

$$q(x_k = d_{id} \mid x_l \in L_i \setminus \{x_k\}) = \frac{1}{Z_\pi} \exp \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})$$

  where $\mathbf{z}_{|S_j}$ is the set of value assignments for the variables in the scope S_j of constraint f_j and Z_π is the normalizing constant. We compute q using $|D_i|$ parallel threads.

To ensure that the procedure returns the best sample found, we verify whether there is an improvement on the best utility (lines 87-88). At the end of the sampling trials, it stores its best sample and utility in the r_{id}-th row in the MVA_TABLE M_i and vector U_i, respectively (line 40).

5.3.3 Theoretical Analysis

We now introduce theoretical properties to relate the quality of DCOP solutions to MCMC sampling strategies, provide bounds on convergence rates for DMCMC algorithms based on MCMC sampling, and provide some complexity analyses of DMCMC requirements. Throughout this section, we assume that the Markov chain $(\mathbf{z}^0, \mathbf{z}^1, \ldots)$ under discussion has finite state space $\mathbf{S}$, a transition matrix T that is irreducible and aperiodic, and has a stationary distribution the posterior π.

Lemma 7. *The expected number of samples $\tau_{\mathbf{z}^*}$ for a MCMC algorithm to get an optimal solution $\mathbf{z}^*$ is*

$$E_{\mathbf{z}^*}(\tau_{\mathbf{z}^*}) = \frac{1}{\pi(\mathbf{z}^*)}.$$

This Lemma is a direct consequence of Lemma 6.

Theorem 12. *The expected number of samples to find an optimal solution $\mathbf{z}^*$ with an MCMC sampling algorithm T is no greater than with a uniform sampling algorithm. In other words,*

$$P_T(\mathbf{z}^*) \geq P_{\text{uni}}(\mathbf{z}^*)$$

Theorem 12 is introduced by Nguyen et al. [84] and can be generalized to any MCMC sampling algorithm that is irreducible and aperiodic as convergence is guaranteed in a finite number of time steps.

Definition 20 (Top α_i-Percentile Solutions)**.** *For an agent a_i the* top α_i-percentile solutions *S_{α_i} is a set containing solutions for the local variables L_i that are no worse than any solution in the supplementary set $D_i \setminus S_{\alpha_i}$, and $\frac{|S_{\alpha_i}|}{|D_i|} = \alpha_i$. Given a list of agents $a_1, \ldots, a_m$, the top $\bar{\alpha}$-percentile solutions $S_{\bar{\alpha}}$ is defined as $S_{\bar{\alpha}} = S_{\alpha_1} \times \ldots \times S_{\alpha_m}$.*

Lemma 8. *After $N_i = \frac{1}{\alpha_i \epsilon_i}$ number of samples with an MCMC sampling algorithm T, the probability that the best solution found thus far $\mathbf{z}_{N_i}$ is in the top α_i for an agent a_i is at least $1 - \epsilon_i$:*

$$P_{\text{T}}\left(\mathbf{z}_{N_i} \in S_{\alpha_i} \mid N_i = \frac{1}{\alpha_i \cdot \epsilon_i}\right) \geq 1 - \epsilon_i.$$

This Lemma is a direct extension of Theorem Theorem 12, introduced in [84].

Theorem 13. *Given m agents $a_1, \ldots, a_m \in \mathbf{A}$, and a number of samples $N_i = \frac{1}{\alpha_i \cdot \epsilon_i}$ $(i = 1, \ldots, m)$, the probability that the best complete solution found thus far $\mathbf{z}_{\mathbf{N}}$ is in the top $\bar{\alpha}$-percentile is greater than or equal to $\prod_{i=1}^{m}(1 - \epsilon_i)$, where $\mathbf{N} = \bigwedge_{i=1}^{m} N_i$. In other words,*

$$P_{\text{T}}\left(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}\right) \geq \prod_{i=1}^{m}(1 - \epsilon_i).$$

Proof. Let $\mathbf{z}_N$ denote the best solution found so far in the process resolution and $\mathbf{z}_{N_i}$ denote the best partial assignment over the variables held by agent a_i found after N_i samples. Let $\mathbf{S}_i$ be a random variable describing whether $\mathbf{z}_{N_i} \in S_{\alpha_i}$. Thus:

$$\begin{aligned}
&P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}) && \text{(5.3a)}\\
&\quad = P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}_1, \dots, \mathbf{N}_m) && \text{(5.3b)}\\
&\quad = P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\alpha_1} \times \ldots \times S_{\alpha_m} \mid \mathbf{N}_1, \dots, \mathbf{N}_m) && \text{(5.3c)}\\
&\quad = P_{\mathrm{T}}(\mathbf{S}_1, \dots, \mathbf{S}_m \mid \mathbf{B}_1, \dots, \mathbf{B}_m, \mathbf{N}_1, \dots, \mathbf{N}_m) && \text{(5.3d)}
\end{aligned}$$

where each $\mathbf{B}_i$ $(i{=}1, \dots, m)$ is a random variable describing a particular value assignment associated to the boundary variables B_i for the agent a_i. They are introduced to relate each of the $\mathbf{z}_{\mathbf{N_i}}$ to each other, which are sampled independently.

Since the values sampled in the local variable of a_i are dependent only of the values of the boundary values B_i, it follows that $\mathbf{S}_i$ is conditionally dependent of $\mathbf{B}_i$ but conditionally independent of all other $\mathbf{B}_j$, with $j \neq i$:

$$S_i \perp\!\!\!\perp B_j \mid B_i$$

for all $j = 1 \dots m$ and $j \neq i$. Noticing that, given random variables a, b, c, whenever $a \perp\!\!\!\perp b \mid c$ we can write: $P(a \mid b, c) = P(a \mid c)$, and that $P(a, b \mid c) = P(a \mid b, c)$, it follows that Equation (5.3d) can be rewritten as:

$$\begin{aligned}
&P_{\mathrm{T}}(\mathbf{S}_1 \mid \mathbf{B}_1, \mathbf{N}_1) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{S}_m \mid \mathbf{B}_m, \mathbf{N}_m)\\
&\quad = P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N_1}} \in S_{\alpha_1} \mid \mathbf{B}, \mathbf{N}) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N_m}} \in S_{\alpha_m} \mid \mathbf{B}, \mathbf{N}) && \text{(5.4a)}\\
&\quad \geq (1 - \epsilon_1) \cdot \ldots \cdot (1 - \epsilon_m) && \text{(5.4b)}\\
&\quad = \prod_{i=1}^{m} (1 - \epsilon_i). && \text{(5.4c)}
\end{aligned}$$

for any of the assignments of the variables in B_i, as the utility functions involving variables in the boundary of any two agents are solved optimally. □

Theorem 14 (Number of Messages)**.** *The number of messages required by DMCMC is linear in the size of the agents.*

Proof. There are $|\mathbf{A}| - 1$ UTIL messages (one through each tree-edge) and $|\mathbf{A}| - 1$ VALUE messages. The DFS construction, like in DPOP, also produces a linear number of messages (usually it requires $2|\mathbf{A}|$ messages). Thus, the total number of messages required is $O(|\mathbf{A}|)$. □

Note that, unlike DPOP, which requires $O(|\mathbf{X}|)$ messages, no message exchange is required to solve the constraints defined over the scope of the local variables each agent, which is achieved via local sampling.

Theorem 15 (Space Requirements)**.** *The memory requirement of each DMCMC agent is exponential in the induced width of the problem.*

Proof. Each agent $a_i \in \mathbf{A}$ needs to store its own utilities and the corresponding solution (value assignment for all non-boundary local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the boundary variables $x_i^k \in B_i$, thus requiring $O(|D_i|^{|B_i|})$ space. Moreover during the UTIL propagation phase, each agent a_i stores the UTIL messages of each of its children $a_c \in C_{a_i}$, which also sends messages of size

$O(|D_i|^{|B_c|})$. Joint and projection operations can be performed efficiently within $O(|D_i|^{N_{S_i}-|B_i|})$ space, where N_{S_i} is the number of variables in the separator set of a_i which is involved in a constraint with some variable in B_i. Thus the memory complexity of each agent is exponential in the *induced width*—the maximum number of boundary variables of the parent of an agent involved in a constraint with the boundary variable of the agent itself. $\square$

Exponential size messages do not represent necessary a limitation. One can bound the maximum message size and serialize big messages by letting the back-edge handlers ask explicitly for solutions and utilities for a subset of their values sequentially. Moreover, one could reduce the exponential memory requirement at cost of sacrificing completeness, and propagating solutions for a bounded set of value combinations instead of all combination of values of the boundary variables. Researchers have investigated some of these approaches for reducing the memory requirement of DPOP [95, 99, 100].

5.3.4 Related Work

To the best of our knowledge, there are only two sampling algorithms developed to solve DCOPs, namely DUCT [88] and Distributed Gibbs [84]. Both algorithms perform repeated sampling trials on the entire space of all variables, where DUCT uses the UCT algorithm [60], which maintains and uses upper confidence bounds on each value of a variable to determine which value to choose during the sampling process, while Distributed Gibbs uses the Gibbs sampling procedure.

In contrast, DMCMC partitions the search space into independent subsets (of local variables of an agent), and performs repeated sampling trials on each of these subsets in parallel. As a result, DMCMC is able to exploit the parallel processes with the use of GPGPUs.

5.3.5 Experimental Evaluation

We implemented CPU and GPU versions of the DMCMC framework with Gibbs (D-Gibbs) and Metropolis-Hastings (D-MH) as the MCMC sampling algorithms. The CPU versions sample sequentially, while the GPU versions sample in parallel with GPGPUs. We compare them against DPOP [96] (an optimal algorithm), MGM and MGM2 [72] (sub-optimal algorithms).[6] We use publicly-available implementations of these algorithms, which are implemented in the FRODO framework [68]. We run our experiments on a Intel(R) Xeon(R) CPU, 2.4GHz, 32GB of RAM, Linux x86_64, equipped with a Tesla C2075, 14SM, 448-core, 1.15 clock rate, CUDA 2.0. We measure runtime using the simulated time metric [111] and perform evaluations on meeting scheduling and smart grid network problems.

Meeting Scheduling Problems: In these problems, meetings need to be scheduled between members of a hierarchical organization, (e.g., employees of a company; students, faculty members, and staff of a university), taking restrictions in their availability as well as their priorities into account. We used the Private Events as Variables (PEAV) problem formulation [73], which is commonly used in the literature. Figure 5.7 show the average (a,c) and the median (b,d) results for 100 runs, together with the standard deviations (vertical bars) of problem instances with a variable number of agents and fixing each agent's

[6] We did not compare against Distributed Gibbs as the authors' implementation does not handle hard constraints, and we do not compare against DUCT as no public implementation is available.

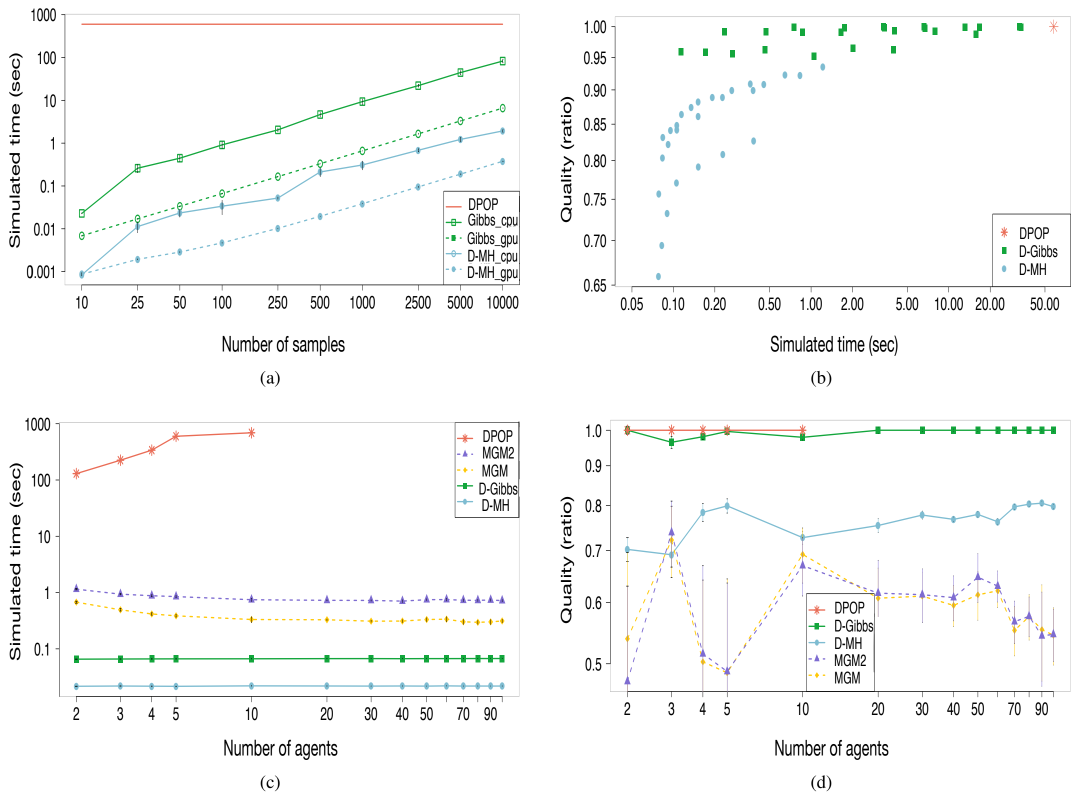


Figure 5.7: Experimental Results: Meeting Scheduling Problems

number of variables to 10, the domain size of its variables to 12, its local constraint graph density to 0.7, and its number of boundary variables to 1.

We first compare the performance of the CPU and GPU DMCMC algorithms on an instance of the meeting scheduling problem with 5 agents. Figure 5.7(a) shows the run-times of the CPU (solid line) and GPU (dotted line) versions of DMCMC together with DPOP (solid horizontal line). The results shows that there is a clear benefit to parallelizing the sampling operations with GPGPUs, exhibiting more than one order of magnitude speed up. In the rest of the experiments, we show the GPU version only. Figure 5.7(b) shows the tradeoff between quality and runtime for the D-Gibbs and D-MH for a range of initial parameters $R = \{1, 10, 50, 100\}$ and $T = \{100, 250, 500, 1000, 5000, 10000\}$. The prediction quality increases with increasing R and T. D-Gibbs is slower than D-MH, as it requires computing normalization constants, which is computationally expensive even when parallelized. However, D-Gibbs finds better solutions.

Finally, we evaluate the algorithms in 14 benchmarks where we vary the number of agents $|\mathbf{A}|$ from 2 to 100. We set $S = 100$ and $R = 10$ for D-Gibbs and $S = 500$ and $R = 100$ for D-MH. Figures 5.7(c) and (d) show the runtime and solution qualities, respectively. DPOP ran out of memory for problems with more than 10 agents. The DMCMC algorithms are up to 2 order of magnitude faster than MGM and MGM2 and can find better solutions, demonstrating the strength of sampling-based approaches over incomplete search algorithms. The results are statistically significant with p-values $< 1.0^{-10}$ for all

Alg.	$\lvert\mathbf{A}\rvert = 100$		$\lvert\mathbf{A}\rvert = 250$		$\lvert\mathbf{A}\rvert = 500$	
D-MH	0.025	(0.01%)	0.026	(0.02%)	0.031	(0.00%)
D-Gibbs	1.387	(1.72%)	1.285	(1.72%)	1.318	(1.71%)
DPOP	15.58	(0.00%)	59.06	(0.00%)	70.01	(0.00%)

Table 5.1: Experimental Results: Smart Grid Networks

parameter configurations.

Grid Networks: We generate grid network problems, which consists of buildings that have power generation and consumption capabilities. Additionally, each building can also send and receive power to and from neighboring buildings. A cost function is associated to the generation and consumption of power of each building. The goal is to minimize the total cost subject to flow conservation and capacity constraints. This problem is called the Comprehensive Customer-Driven Microgrid Optimization Problem in the literature [46].

As the problem definition does not define the network topology, we used clustered scale-free graphs [112], where each cluster has a few high density nodes. We generated problem instances where we vary the number of agents $|\mathbf{A}| = \{100, 250, 500\}$ and the number of local variables of each agent depends on the number of neighboring agents. We fix the domain sizes to 11 and the maximum constraint arity to 5. Table 5.1 reports the simulated run-times (in seconds) and the error in solution quality (in parenthesis). These results show that the DMCMC algorithms can find close-to-optimal solutions significantly faster than DPOP. We omit MGM and MGM2 as they always found unsatisfactory solutions due to the large number of hard constraints in the problem.

5.4 Summary

In this chapter, we presented an investigation on the use of GPGPUs to exploit SIMT-style parallelism from DP-based methods to solve COPs and DCOPs, and from MCMC sampling algorithms within the MVA decomposition framework to solve DCOPs. We proposed a procedure, inspired by BE (for COPs) and DPOP (for DCOPs), that makes use of multiple threads to parallelize the aggregation and projection phases of the DP-based algorithms. Our experimental results show that the use of GPGPUs may provide significant advantages in terms of runtime and scalability. Furthermore, motivated by (*i*) the assumption in most DCOP algorithms that each agents owns exactly one variable; (*ii*) the recent introduction of sampling-based DCOP algorithms, which have been shown to outperform existing incomplete DCOP algorithms; and (*iii*) the advances in General Purpose Graphical Processing Units (GPGPUs), we introduced the Distributed MCMC framework. Such framework uses the MVA decomposition (see Chapter 3) to solve the general DCOP, using a DPOP-based algorithm, and decomposes the DCOP into independent sub-problems that can each be sampled in parallel exploiting GPGPUs. Our experimental results show that it can find near-optimal solutions up to one order of magnitude faster than MGM and MGM2.

The proposed results are significant—the wide availability of GPGPUs provides access to parallel computing solutions that can be used to improve efficiency of (D)COP solvers. Furthermore, GPGPUs are renowned for their complex architectures (multiple memory levels with very different size and speed characteristics; relatively slow cores), which often create challenges to the effective exploitation of parallelism from irregular applications; the strong experimental results indicate that the proposed algorithms are well-suited to GPGPU architectures.

Therefore, these results validate the hypothesis that one can exploit highly parallel computational models to enhance current DCOP solution techniques, which is exciting as GPGPUs provide access to hundreds of computing cores at a very affordable cost.

6 Conclusions

Distributed Constraint Optimization Problems (DCOPs) have emerged as a popular formalism for distributed reasoning and coordination in Multi-Agent System where several agents cooperate to optimize a global cost function. They represent a powerful approach to the description and resolution of many practical problems, and serve several applications such as distributed scheduling, coordination of unmanned air vehicles, smart grid electric networks, and sensor networks. Typical real world applications are characterized by complex dynamics and interactions among a large number of entities, which translate into hard combinatorial problems, posing significant challenges from a computational point of view.

In this dissertation we identified two major challenges in applying DCOPs algorithms to large complex problems: (1) *Modeling assumptions*: as current resolution methods detach the model from the resolution process, imposing limiting assumptions on the capabilities of an agent, and (2) *Solving capabilities*: as the inability of current approaches to capitalize on the presence of structural information which may allow incoherent/unnecessary data to reticulate among the agents as well as to exploit latent structure of the agent's local problems, and/or of the problem of interest.

This dissertation has focused on addressing such challenges by investigating the hypothesis that one can exploit the latent structure of DCOPs in both problem modeling and problem resolution phases, and using GPGPU-level parallelism. We briefly review below each of these contributions, and outline potential directions for future work.

6.1 Exploiting the Structure of DCOPs from Problem Modeling

We began our path by noticing that most DCOP resolution approaches are designed following the underlying assumption that each agent controls exclusively a single variable of the problem. However, modeling many real-world complex applications, requires each agent to solve complex problems, and to control a large number of variables. We reviewed two *reformulation* techniques that are commonly adopted to address this modeling assumption, and argued that such techniques could be arbitrarily inefficient, as they ignore the structure present in the problem model. We thus proposed a *Multi-Variable Agent (MVA)* DCOP decomposition technique which exploits co-locality of each agent's variables, allowing us to adopt efficient centralized techniques within each DCOP agent. Crucially, such decomposition preserves agent privacy. The advantages of using the MVA decomposition were demonstrated by our experimental results, showing remarkable improvements in terms of network load and scalability, outperforming several classes of non-decomposed DCOP algorithms.

Potential Directions for Future Work

The MVA decomposition defines a clear separation between the distributed agent coordination and the centralized agent subproblem resolution. This separation allows the use of efficient centralized solvers to solve agent subproblems as well as the use of potentially different solvers for different agents, each designed to exploit domain-specific properties. Thus, in the future we plan to investigate the integration of several DCOP algorithms with efficient centralized optimization solvers (such as, CPLEX [20], Gurobi [87]) or *Constraint Programming* solvers (such as, Gecode [113]), dedicated to the resolution of the agent subproblems. In addition we plan to investigate the application of propagation schemes (e.g., as in [35]) to further reduce agent-to-agent communication.

We plan to apply the proposed integrated solution to solve smart building scheduling problems within a micro-grid. In such problems several buildings, each modeled by an agent, need to schedule the execution of their appliances, and are subjected to a maximal amount of energy that can be used at each time of the day, as well as exposed to price variations of the energy consumed in different hours of the day.

6.2 Exploiting the Structure of DCOPs during Problem Solving

Next, we investigated solutions to boost DCOP solving capabilities. We identified two orthogonal directions to exploit the structure of DCOPs during problem solving. The first solution focused on exploiting the hard constraint of the problem, and we proposed *Branch Consistency (BrC)*, a type of consistency that applies to paths in pseudo-trees aimed to prune the search space and to reduce the size of the messages exchanged among agents. We proved that such form of consistency enforces a more effective pruning than those based on domain consistency, and we applied BrC to reduce the space explored by DPOP agents, one of the most competitive DCOP algorithms. The resulting algorithm, BrC-DPOP, was shown to effectively exploit the information encoded in the hard constraints, substantially reducing the network load and the resolution time compared to other complete algorithms, and without incurring to any additional privacy loss. The second solution focused on exploiting problem structure from domain-dependent knowledge and it suitable to tackle large problems which cannot be coped with complete DCOP approaches. Such solution resulted in the Distributed Large Neighboring Search (D-LNS), a local search framework for DCOPs which builds on the strengths of centralized Large Neighboring Search, which iteratively explores complex neighborhoods of the search space to find better candidate solutions. The resulting framework has several qualities: It provides quality guarantees by refining both upper and lower bounds of the solution found during the iterative process; It is anytime; and it inherently uses insights from the CP techniques to take advantage on the presence of hard constraints. Our experimental analysis showed that D-LNS based algorithms converge faster to better solutions, compared to other incomplete algorithms, and provide tighter solution quality bounds.

Potential Directions for Future Work

Our plan for future work is to extend BrC-DPOP to handle higher arity constraints. This can be done by substituting the VRM structures with either consistency graphs or higher dimension VRMs. We suspect that there will be a tradeoff between runtime and memory requirement between the two approaches, where using higher dimension VRMs is faster but uses more memory. We also plan to extend BrC-DPOP to memory-bounded versions similar to MB-DPOP [99] in order to scale to even larger problems. Finally, we plan to explore propagation of soft constraints similar to the versions of BnB-ADOPT with soft AC enforcement [9, 49, 47].

On the D-LNS side, we plan to investigate other schemes to incorporate into the repair phase of D-LNS (e.g., propagation techniques [9, 35, 47] to better prune the search space) that actively exploit the bounds reported during the iterative procedure, as well as the use of General Purpose Graphics Processing Units to parallelize the search for better speedups [16, 34]) in presence of large agent's local subproblems. We strongly believe that this framework has the potential to solve very large distributed constraint optimization problems, with thousands of agents, variables, and constraints, and we plan a systematic evaluation for the near future.

6.3 Exploiting the use of Accelerated Hardware in DCOP resolution

Motivated by the large interest in DP-based DCOP algorithms within the AAMAS community (see e.g., [99, 101, 63, 79]), we also investigated the use of GPGPU-based solutions to enhance the efficiency of such approaches. Indeed, the structure exploited by DP-based approaches in constructing solutions makes it suitable to exploit the SIMT paradigm, which is widely used in modern general purpose graphic processing units. Thus, we proposed a DP-based algorithm that exploits parallel computation using GPGPUs to solve DCOPs. Our proposal employs GPGPU hardware to speed up the inference process of DP-based methods, representing an alternative way to enhance the performance of DP-based constraint optimization approaches. Our results show significant improvements in performance and scalability over other state-of-the-art DP-based solutions, with speedup up to two order of magnitude.

The explicit separation between the DCOP resolution process and the centralized agent problem, enabled by our MVA DCOP decomposition, capacitate agents to solve their local problem trough a variety of techniques. Motivated by the high complexity of the agent local problem, we proposed the use of hierarchical parallel models, where each agent can *(*1) solve its local problem independently from those of other agents, and *(*2) parallelize the computations within its own local problem. We thus introduced a framework to solve independent local problems, in parallel, using sampling-based algorithms, harnessing the multitude of computational units offered by GPGPUs. This approach led to significant improvements in the runtime of the algorithm resolution.

Potential Directions for Future Work

In the future we plan to extend these GPGPU-based frameworks to reduce their memory requirements, in a way similar to what proposed in MB-DPOP [99] and PC-DPOP [100].

While envisioning further research in this area, we anticipate several challenges: In terms of implementation, GPGPU programming can be more demanding when compared to a classical sequential implementation. One of the current limitations for (D)COP-based GPGPU approaches is the absence of solid abstractions that allow component integration, modularly, without restructuring the whole program.

Exploiting the integration of CPU and GPGPU computations is a key factor to obtain competitive solvers performance. Complex and repeated calculations should be delegated to GPGPUs, while simpler and memory intensive operations should be assigned to CPUs. It is however unclear how to determine good tradeoffs of such integrations. For instance, repeatedly invoking many memory demanding GPGPU kernels could be detrimental to the overall performance, due to the high cost of allocating the device memory (e.g., shared memory). Creating lightweight communication mechanisms between CPU and GPGPU (for instance, by taking advantage of the asynchronism of CUDA streams) to allow active GPGPU kernels to be used in multiple instances could be a possible solution to investigate.

Bibliography

[1] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[2] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] Alejandro Arbelaez and Philippe Codognet. A GPU Implementation of Parallel Constraint-based Local Search. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP)*, pages 648–655, 2014.

[4] James Atlas and Keith Decker. Coordination for Uncertain Outcomes using Distributed Neighbor Exchange. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1047–1054, 2010.

[5] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] C. Bessiere and J.C. Regin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 309–315, 2001.

[7] Christian Bessiere. Constraint propagation. *Handbook of Constraint Programming*, pages 29–83, 2006.

[8] Christian Bessiere, Ismel Brito, Patricia Gutierrez, and Pedro Meseguer. Global Constraints in Distributed Constraint Satisfaction and Optimization. *Computer Journal*, 57(6):906–923, 2014.

[9] Christian Bessiere, Patricia Gutierrez, and Pedro Meseguer. Including Soft Global Constraints in DCOPs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 175–190, 2012.

[10] Emma Bowring, Milind Tambe, and Makoto Yokoo. Multiply-constrained distributed constraint optimization. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1413–1420. ACM, 2006.

[11] Stephen Boyd and Jacob Mattingley. Branch and Bound Methods. *Notes for EE364b, Stanford University*, pages 2006–07, 2007.

[12] Vincent Boyer, Didier El Baz, and Moussa Elkihel. Solving Knapsack Problems on GPU. *Computers & Operations Research*, 39(1):42–47, 2012.

[13] Ismel Brito and Pedro Meseguer. Improving DPOP with function filtering. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 141–158, 2010.

[14] David Burke and Kenneth Brown. Efficiently Handling Complex Local Problems in Distributed Constraint Optimisation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 701–702, 2006.

[15] Bertrand Cabon, Simon De Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

[16] Federico Campeotto, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 189–194, 2014.

[17] Federico Campeotto, Alessandro Dal Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the Use of GPUs in Constraint Solving. In *Proceedings of the Practical Aspects of Declarative Languages (PADL)*, pages 152–167, 2014.

[18] Imen Chakroun, Mohand-Said Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing Thread Divergence in a GPU-accelerated Branch-and-Bound Algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[19] Martin Cooper and Thomas Schiex. Arc Consistency for Soft Constraints. *Artificial Intelligence*, 154(1):199–227, 2004.

[20] ILOG Cplex. 11.0 User's manual. *ILOG SA, Gentilly, France*, page 32, 2007.

[21] Alessandro Dal Palu, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@ SAT: SAT Solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.

[22] George Dantzig and Mukund Thapa. *Linear Programming 1: Introduction*. Springer-Verlag, 1997.

[23] George Dantzig and Mukund Thapa. *Linear Programming 2: Theory and Extensions*. Springer-Verlag, 2003.

[24] John Davin and Pragnesh Modi. Hierarchical Variable Ordering for Multiagent Agreement Problems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1433–1435, 2006.

[25] Rina Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In *Learning in graphical models*, pages 75–104. Springer, 1998.

[26] Rina Dechter, editor. *Constraint Processing*. Morgan Kaufmann, 2003.

[27] Gregory Frederick Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488, New York, NY, 2011. ACM Press.

[28] Edsger W Dijkstra. Self-stabilization in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[29] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. In *Proceedings of the fourteenth annual ACM Symposium on Principles of Distributed Computing*, page 255. ACM, 1995.

[30] P. Erdös and A. Rényi. On Random Graphs I. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

[31] Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas Jennings. Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 639–646, 2008.

[32] Ferdinando Fioretto, Federico Campeotto, Agostino Dovier, Enrico Pontelli, and William Yeoh. Large Neighborhood Search with Quality Guarantees for Distributed Constraint Optimization Problems. In *AAMAS*, pages 1835–1836, 2015.

[33] Ferdinando Fioretto, Federico Campeotto, Luca Da Rin Fioretto, William Yeoh, and Enrico Pontelli. GD-Gibbs: A GPU-based Sampling Algorithm for Solving Distributed Constraint Optimization Problems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1339–1340, 2014.

[34] Ferdinando Fioretto, Tiep Le, Enrico Pontelli, William Yeoh, and Tran Cao Son. Exploiting GPUs in Solving (Distributed) Constraint Optimization Problems with Dynamic Programming. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 121–139, 2015.

[35] Ferdinando Fioretto, Tiep Le, William Yeoh, Enrico Pontelli, and Tran Cao Son. Improving DPOP with Branch Consistency for Solving Distributed Constraint Optimization Problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 307–323, 2014.

[36] Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. Decomposition Techniques for DCOPs to Exploit Multi-Variable Agents and Multi-Level Parallelism. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1823–1824, 2015.

[37] Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. Multi-Variable Agents Decompositions for DCOPs to Exploit Multi-Level Parallelism. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, page (in press), 2016.

[38] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

[39] Amir Gershman, Amnon Meisels, and Roie Zivan. Asynchronous Forward-Bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, 2009.

[40] Amir Globerson and Tommi Jaakkola. Fixing Max-Product: Convergent Message Passing Algorithms for MAP LP-Relaxations. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, pages 553–560, 2007.

[41] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized Large Neighborhood Search for Cumulative Scheduling. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 5, pages 81–89, 2005.

[42] Rachel Greenstadt, Barbara Grosz, and Michael Smith. SSDPOP: Improving the Privacy of DCOP with Secret Sharing. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1098–1100, 2007.

[43] Rachel Greenstadt, Jonathan Pearce, and Milind Tambe. Analysis of Privacy Loss in DCOP Algorithms. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 647–653, 2006.

[44] Tal Grinshpoun, Alon Grubshtein, Roie Zivan, Arnon Netzer, and Amnon Meisels. Asymmetric Distributed Constraint Optimization Problems. *Journal of Artificial Intelligence Research*, 47:613–647, 2013.

[45] Tal Grinshpoun and Amnon Meisels. Completeness and Performance Of The APO Algorithm. *Journal of Artificial Intelligence Research*, 33:223–258, 2008.

[46] Saurabh Gupta, Palak Jain, William Yeoh, S Ranade, and Enrico Pontelli. Solving customer-driven microgrid optimization problems as DCOPs. In *International Workshop on Distributed Constraint Reasoning (DCR)*, pages 45–59, 2013.

[47] Patricia Gutierrez, Jimmy Lee, Ka Man Lei, Terrence Mak, and Pedro Meseguer. Maintaining Soft Arc Consistencies in BnB-ADOPT$^+$ during Search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 365–380, 2013.

[48] Patricia Gutierrez and Pedro Meseguer. Saving Redundant Messages in BnB-ADOPT. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1259–1260, 2010.

[49] Patricia Gutierrez and Pedro Meseguer. Improving BnB-ADOPT$^+$-AC. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 273–280, 2012.

[50] Patricia Gutierrez and Pedro Meseguer. Removing Redundant Messages in n-ary BnB-ADOPT. *Journal of Artificial Intelligence Research*, 45:287–304, 2012.

[51] Patricia Gutierrez, Pedro Meseguer, and William Yeoh. Generalizing ADOPT and BnB-ADOPT. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 554–559, 2011.

[52] Tianyi David Han and Tarek S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 3:1–3:8, New York, NY, 2011. ACM Press.

[53] W Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[54] Daisuke Hatano and Katsutoshi Hirayama. DeQED: An Efficient Divide-and-Coordinate Algorithm for DCOP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 566–572, 2013.

[55] Pascal Van Hentenryck and Laurent Michel. *Constraint-based Local Search*. The MIT Press, 2009.

[56] Katsutoshi Hirayama and Makoto Yokoo. Distributed Partial Constraint Satisfaction Problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 222–236, 1997.

[57] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to NP-Completeness of Knapsack Problems. In *Knapsack Problems*, pages 483–493. Springer Berlin Heidelberg, 2004.

[58] Christopher Kiekintveld, Zhengyu Yin, Atul Kumar, and Milind Tambe. Asynchronous Algorithms for Approximate Distributed Constraint Optimization with Quality Bounds. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 133–140, 2010.

[59] Yoonheui Kim and Victor Lesser. Improved Max-Sum Algorithm for DCOP with n-ary Constraints. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 191–198, 2013.

[60] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[61] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[62] Akshat Kumar, Boi Faltings, and Adrian Petcu. Distributed Constraint Optimization with Structured Resource Constraints. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 923–930, 2009.

[63] Akshat Kumar, Adrian Petcu, and Boi Faltings. H-DPOP: Using Hard Constraints for Search Space Pruning in DCOP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 325–330, 2008.

[64] Akshat Kumar and Shlomo Zilberstein. MAP Estimation for Graphical Models by Likelihood Maximization. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, pages 1180–1188, 2010.

[65] Mohamed Esseghir Lalami, Didier El Baz, and Vincent Boyer. Multi GPU implementation of the simplex algorithm. In *Proceedings of the International Conference on High Performance Computing and Communication (HPCC)*, volume 11, pages 179–186, 2011.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[67] Tiep Le, Tran Cao Son, Enrico Pontelli, and William Yeoh. Solving Distributed Constraint Optimization Problems with Logic Programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1174–1181, 2015.

[68] Thomas Léauté and Boi Faltings. E [DPOP]: Distributed Constraint Optimization under Stochastic Uncertainty using Collaborative Sampling. In *International Workshop on Distributed Constraint Reasoning (DCR)*, pages 87–101, 2009.

[69] Thomas Léauté and Boi Faltings. Distributed Constraint Optimization Under Stochastic Uncertainty. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 11, pages 68–73, 2011.

[70] Alan Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[71] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25(1):65–74, January 1985.

[72] Rajiv Maheswaran, Jonathan Pearce, and Milind Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.

[73] Rajiv Maheswaran, Milind Tambe, Emma Bowring, Jonathan Pearce, and Pradeep Varakantham. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Event Scheduling. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 310–317, 2004.

[74] Roger Mailler and Victor Lesser. Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 438–445, 2004.

[75] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[76] Toshihiro Matsui, Hiroshi Matsuo, Marius Silaghi, Katsutoshi Hirayama, and Makoto Yokoo. Resource Constrained Distributed Constraint Optimization with Virtual Variables. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 120–125, 2008.

[77] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087, 1953.

[78] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Berlin Heidelberg, 1999.

[79] Sam Miller, Sarvapali D Ramchurn, and Alex Rogers. Optimal Decentralised Dispatch of Embedded Generation in the Smart Grid. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 281–288, 2012.

[80] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[81] Pragnesh Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.

[82] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[83] Duc Thien Nguyen, William Yeoh, and Hoong Chuin Lau. Stochastic Dominance in Stochastic DCOPs for Risk-sensitive Applications. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 257–264, 2012.

[84] Duc Thien Nguyen, William Yeoh, and Hoong Chuin Lau. Distributed Gibbs: A memory-bounded sampling-based DCOP algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 167–174, 2013.

[85] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 1984.

[86] Tenda Okimoto, Yongjoon Joe, Atsushi Iwasaki, Makoto Yokoo, and Boi Faltings. Pseudo-tree-based Incomplete Algorithm for Distributed Constraint Optimization with Quality Bounds. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 660–674, 2011.

[87] Gurobi Optimization. Inc. gurobi optimizer reference manual, 2014.

[88] Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 528–534, 2012.

[89] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[90] Krzysztof Pawłowski, Karol Kurach, Tomasz Michalak, and Talal Rahwan. Coalition structure generation with the graphic processor unit. Technical Report CS-RR-13-07, Department of Computer Science, University of Oxford, 2104.

[91] Jonathan Pearce and Milind Tambe. Quality Guarantees on k-Optimal Solutions for Distributed Constraint Optimization Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1446–1451, 2007.

[92] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[93] Federico Pecora, P Modi, and Paul Scerri. Reasoning About and Dynamically Posting n-ary Constraints in ADOPT. In *International Workshop on Distributed Constraint Reasoning (DCR)*, volume 7, 2006.

[94] Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 482–495, 2004.

[95] Adrian Petcu and Boi Faltings. Approximations in Distributed Optimization. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 802–806, 2005.

[96] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1413–1420, 2005.

[97] Adrian Petcu and Boi Faltings. ODPOP: An Algorithm for Open/Distributed Constraint Optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 703–708, 2006.

[98] Adrian Petcu and Boi Faltings. A Hybrid of Inference and Local Search for Distributed Combinatorial Optimization. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT)*, pages 342–348, 2007.

[99] Adrian Petcu and Boi Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1452–1457, 2007.

[100] Adrian Petcu, Boi Faltings, and Roger Mailler. PC-DPOP: A New Partial Centralization Algorithm for Distributed Optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 167–172, 2007.

[101] Adrian Petcu, Boi Faltings, and David Parkes. M-DPOP: Faithful Distributed Implementation of Efficient Social Choice Problems. *Journal of Artificial Intelligence Research*, 32:705–755, 2008.

[102] Claude-Guy Quimper and Toby Walsh. Global Grammar Constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 751–755. Springer, 2006.

[103] Gareth O Roberts and Adrian FM Smith. Simple conditions for the convergence of the Gibbs sampler and Metropolis-Hastings algorithms. *Stochastic Processes and Their Applications*, 49(2):207–216, 1994.

[104] Alex Rogers, Alessandro Farinelli, Ruben Stranders, and Nicholas Jennings. Bounded Approximate Decentralised Coordination via the Max-Sum Algorithm. *Artificial Intelligence*, 175(2):730–759, 2011.

[105] Emma Rollon and Javier Larrosa. Improved Bounded Max-Sum for Distributed Constraint Optimization. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 624–632, 2012.

[106] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[107] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[108] J. Sanders and E. Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.

[109] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 417–431, 1998.

[110] Ruben Stranders, Francesco Maria Delle Fave, Alex Rogers, and Nick Jennings. U-gdl: A decentralised algorithm for dcops with uncertainty. Technical report, University of Southampton, Department of Electronics and Computer Science, 2011.

[111] Evan Sultanik, Pragnesh Jay Modi, and William C Regli. On modeling multiagent task scheduling as a distributed constraint optimization problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1531–1536, 2007.

[112] Wai M Tam, Francis CM Lau, and CK Tse. Construction of scale-free networks with adjustable clustering. In *Proceedings of the International Symposium on Nonlinear Theory and its Applications*, pages 257–260, 2008.

[113] Gecode Team. Gecode: Generic constraint development environment, 2008.

[114] Michael A Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1-4):73–84, 2003.

[115] Gérard Verfaillie and Narendra Jussien. Constraint Solving in Uncertain and Dynamic Environments: A survey. *Constraints*, 10(3):253–281, 2005.

[116] Meritxell Vinyals, Marc Pujol, Juan A Rodriguez-Aguilar, and Jesus Cerquides. Divide-and-coordinate: DCOPs by agreement. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 149–156, 2010.

[117] Meritxell Vinyals, Eric Shieh, Jesús Cerquides, Juan Rodriguez-Aguilar, Zhengyu Yin, Milind Tambe, and Emma Bowring. Quality Guarantees for Region Optimal DCOP algorithms. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 133–140, 2011.

[118] Martin Wainwright and Michael Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 1:1–305, 2008.

[119] Yonghong Wang, Katia Sycara, and Paul Scerri. Towards an Understanding of the Value of Cooperation in uncertain world. In *Proceedings of the International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT)*, volume 2, pages 212–215. IEEE/WIC/ACM, 2011.

[120] William Yeoh. *Speeding Up Distributed Constraint Optimization Search Algorithms*. PhD thesis, University of Southern California, Los Angeles (United States), 2010.

[121] William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

[122] William Yeoh, Xiaoxun Sun, and Sven Koenig. Trading Off Solution Quality for Faster Computation in DCOP Search Algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 354–360, 2009.

[123] William Yeoh, Pradeep Varakantham, and Sven Koenig. Caching Schemes for DCOP Search Algorithms. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 609–616, 2009.

[124] William Yeoh and Makoto Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.

[125] Makoto Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer Berlin Heidelberg, 2001.

[126] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenberg. Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1–2):55–87, 2005.

[127] Neng-Fa Zhou, Roman Bartak, and Agostino Dovier. Planning as tabled logic programming. *Theory and Practice of Logic Programming*, 15(4-5):543–558, 2015.

[128] Neng-Fa Zhou and Agostino Dovier. A tabled Prolog program for solving Sokoban. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, pages 896–897. IEEE, 2011.

[129] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.

[130] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. From Dynamic Programming to Planning. In *Constraint Solving and Planning with Picat*, pages 101–113. Springer, 2015.

[131] Roie Zivan, Robin Glinton, and Katia Sycara. Distributed constraint optimization for large teams of mobile sensing agents. In *Proceedings of the International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT)*, pages 347–354. IEEE/WIC/ACM, 2009.

[132] Roie Zivan, Steven Okamoto, and Hilla Peled. Explorative Anytime Local Search for Distributed Constraint Optimization. *Artificial Intelligence*, 212:1–26, 2014.

[133] Roie Zivan and Hilla Peled. Max/min-sum distributed constraint optimization through value propagation on an alternating DAG. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 265–272, 2012.

A

List of Key Symbols

To facilitate the reading of this disserrtation, we have provided in Table A.1 a summary of the most commonly used notations.

List of key symbols			
a_i	Agent	$\pi(\cdot)$	Projection operator
x_i	Decision variable	$p_i(\cdot)$	Probability function
r_i	Random variable	L_{a_i}	a_i's local variables
D_i	Domain of x_i	N_{a_i}	a_i's neighbors
Ω_i	Event space of r_i	C_{a_i}	a_i's children
f_i	Reward function	PC_{a_i}	a_i's pseudo-children
$\mathbf{x}^i$	Scope of f_i	P_{a_i}	a_i's parent
m	Number of agents	PP_{a_i}	a_i's pseudo-parents
n	Number of variables	$\alpha(f_i)$	agents whose variables are in $\mathbf{x}^i$
q	Number of random variables	E_C	Set of edges of the constraint graph
k	Number of reward functions	E_T	Tree edges of the pseudo-tree
$\mathbf{F}_g$	Global objective function	E_F	Set of edges of the factor graph
$\vec{\mathbf{F}}$	Vector of objective functions	w^*	Induced width of the pseudo-tree
F_i	Objective function in $\vec{\mathbf{F}}$	d	Size of the largest domain
$\vec{\mathbf{F}}^\circ$	Utopia point	l	Size of the largest neighborhood
$\perp$	Infeasible value	z	Size of the largest local variable set
σ	Complete solution	s	Maximal sample size
$\sigma_{\mathbf{x}^i}$	Partial solution of scope $\mathbf{x}^i$ in σ	p	Size of the Pareto set
Σ	State space	b	Size of the largest bin

Table A.1: Commonly Used Symbols and Notations