UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Speeding Up GDL-Based Distributed Constraint Optimization Algorithms in Cooperative Multi-Agent Systems

by

Md. Mosaddek Khan

A thesis submitted in partial fulfilment for the degree of Doctor of Philosophy

April 2018

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

SPEEDING UP GDL-BASED DISTRIBUTED CONSTRAINT OPTIMIZATION ALGORITHMS IN COOPERATIVE MULTI-AGENT SYSTEMS

by Md. Mosaddek Khan

Coping with an increasing number of agents, tasks and/or resources in a complex environment poses an onerous challenge for coordination algorithms that are developed to process constraints in multi-agent systems. In particular, Distributed Constraint Optimization Problems (DCOPs) are a widely studied constraint handling framework for coordinating interactions in cooperative multi-agent systems. For the past decade, a number of algorithms have been developed to solve DCOPs, and they have been applied to many real world applications. However, it is often observed that the outcome obtained from such algorithms becomes outdated or unusable as the optimization process takes too much time. The issue of taking too long to complete the internal operation of a DCOP algorithm is even more severe and commonplace as the system becomes larger. This, in turn, limits the practical scalability of such algorithms. In effect, an optimization algorithm can eventually handle larger systems if the completion time can be minimized. However, it is difficult to maintain the quality of solution and generic applicability whilst minimizing the completion time.

In this thesis, we investigate techniques that have been used to solve DCOPs and examine their efficacy in light of the above mentioned observation. Specifically, we identify that Generalized Distributive Law (GDL) based inference algorithms have a number of axiomatic benefits, and as such, are suited to deploy in practical multi-agent settings. However, scalability remains a widely acknowledged challenge for these algorithms owing to a number of potentially expensive phases. In the multi-agent systems literature, several attempts have sought to improve the scalability of GDL-based algorithms by typically speeding up one of the expensive phases of existing approaches. However, most of them focus on a specific application domain, and therefore cannot be applied to general DCOP settings. Although a few studies have been conducted to speed-up GDL-based algorithms for general settings, they typically experience lack of consistency in their performance.

Against this background, the central problem that this thesis aims to address is of speeding up GDL-based DCOP algorithms, so that they can be applied to general DCOP settings without compromising on solution quality. To accomplish this objective, we determine three of the expensive phases of such algorithms, then speed them up independently. Firstly, the maximization operation – which a GDL-based algorithm performs repetitively during its optimization process. Notably, each of these operates on a search space that grows exponentially with either, or both, of the corresponding constraint function's arity and its associated variables' domain size. Consequently, this particular phase has been considered as one of the main reasons GDL-based algorithms

can be computationally infeasible in practice, which eventually incurs delay in producing the final outcome of these algorithms. To overcome this challenge, we develop a generic domain pruning technique so that the corresponding maximization operator can act upon a significantly reduced search space of 33% to 81%. Moreover, we theoretically prove that the pruned search space obtained by our approach does not affect the outcome of the algorithms.

Secondly, GDL-based algorithms follow the Standard Message Passing (SMP) protocol to exchange messages among the nodes of a corresponding graphical representation of a DCOP. We identify that this incurs a signicant delay in the form of average waiting time for agents to attain the ultimate outcome. Building on this insight, we advance the state-of-the-art by developing a new way of speeding up GDL-based message passing algorithms. In particular, we propose a new cluster-based generic message passing protocol that minimizes the completion time of GDL-based algorithms by replacing the SMP protocol. To elaborate further, our approach utilizes partial decentralization and combines clustering with domain pruning. It also uses a regression method to determine the appropriate number of clusters for a given scenario. We empirically evaluate the performance of our proposed method in different possible settings, and find that it brings down the completion time by around $37 - 85\%$ ($1.6 - 6.5$ times faster) for $100 - 900$ nodes and by around $47 - 91\%$ ($1.9 - 11$ times faster) for $3000 - 10000$ nodes, compared to the current state-of-the-art.

Finally, the conventional DCOP model assumes that the sub-problem that each agent is responsible for (i.e. the mapping of nodes in the constraint graph to agents) is part of the model description. While this assumption is often reasonable, there are many applications where there is some flexibility in making this assignment. Specifically, we recognise that a poor mapping can increase an algorithm's completion time in a significant manner, and that finding an optimal mapping is an NP-hard problem. In the wake of this trade-off, we propose a new time-efficient heuristic to determine a near-optimal mapping of nodes to the participating agents of a DCOP. As a pre-processing step, it works prior to executing the optimization process of a GDL-based algorithm, and can be executed in a centralized or a decentralized manner, depending on the applications' suitability. We empirically demonstrate that it performs at a level of around $90\% - 100\%$ of the optimal mapping. Our results also show a speed-up of $16\% - 40\%$ when compared with the state-of-the-art. This means that a GDL-based algorithm can perform $1.2 - 1.7$ times faster when using node-to-agent mapping obtained by our method. When taken together, the contributions presented in this thesis signify advancement in the state-of-the art of GDL-based DCOP algorithms, in terms of their scalability and applicability, by speeding up their optimization process.

Contents

List of Figures

List of Tables

List of Algorithms

Declaration of Authorship

I, Md. Mosaddek Khan, declare that the thesis entitled *Speeding Up GDL-Based Distributed Constraint Optimization Algorithms in Cooperative Multi-Agent Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published in a number of conference and journal papers (see Section 1.2 for a list).

Signed: .....

Date: .....

Acknowledgements

I would like to thank everyone who supported and encouraged me during my PhD. First and foremost, I want to thank my supervisors, Nicholas R. Jennings and Long Tran-Thanh, for their relentless support, encouragement, guidance and expertise. Thanks a lot for dedicating a tremendous amount of time during our weekly meetings, as well as in correcting my writings. Finally, I am incredibly grateful for the opportunity they have provided me to pursue my PhD at the School of Electronics and Computer Science in University of Southampton. I would like to thank my co-authors, William Yeoh and Sarvapali Ramchurn, for their invaluable technical guidance. I would also like to thank my examiners, Timothy J. Norman and Alessandro Farinelli, for making my viva and thesis finalisation an enriching experience.

Thanks to everybody from the AIC lab who have extended their support to keep me stay on track, and also for providing necessary distractions and different perspectives, during various phases of my PhD. I owe special thanks to Olabambo Oluwasuji, Zoltan Beck, Nhat Truong, Ibrahem Almansour, Radu Pruna, Alper Turan, Alexandry Augustin, Edoardo Manino and Henry Truong Ngoc Cuong for being so helpful.

I gratefully acknowledge the financial support provided by ECS, University of Southampton. I also acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of some of my experiments.

Last but not the least, I have received enormous support from my family. My parents and my two sisters have always given me love and encouragement. Specially, I would not be in this phase of my academic life without my father's stimulation and my mother's sacrifice. Thanks to my lovely wife Nishe for tolerating me during this journey, and for picking me up when I am down. I would be lost without you. To my beloved two-year-old daughter Nameera, thanks for being such a good girl – always cheering me up. Above all, I thank my almighty Allah for letting me through all the difficulties.

Nomenclature

A	A set of cooperating agents in a DCOP
A_i	A single agent in A
X	A set of variables in a DCOP
x_i	A single variable in X
D	The set of discrete and finite variable domains
D_i	The domain of variable x_i
d_i	Domain size of variable x_i
d	Domain size of all the variables when $d_i - d'_i = 0$ for all d_i and d'_i
F	The set of constraint functions/factors in a DCOP
F_i	A single constraint function in F
$\mathbf{x}_i$	The subset of X associated with the function F_i
$\mathbb{S}$	The set of states corresponding to the domains of $\mathbf{x}_j$
$\mathbf{s_i}$	A single state in $\mathbb{S}$
G	A constraint graph corresponding to a DCOP
G_j	A sub-graph of G
F_G	A factor graph representation of G
η	The set of variable (and function) nodes in G or F_G
n	The arity of a function node F_i in F_G (i.e. $n = \lvert\mathbf{x}_i\rvert$)
δ	The node-to-agent mapping function within a DCOP
U, V	Cliques of a junction tree
$\mathcal{K}$	A set stands for a commutative semiring
$Q_{x_i \to F_j}(x_i)$	A variable to function message in the GDL framework
$R_{F_j \to x_i}(x_i)$	A function to variable message in the GDL framework
$Z_i(x_i)$	Generated local objective function in the GDL framework
w_{ij}	The weight that defines the maximum impact of an edge between x_j and F_i in a factor graph
V^*	The optimal solution from a factor graph
$\tilde{V}$	The approximate solution obtained from the transformed acyclic factor graph
t_j	A single task in the formulation of task allocation used by FMS
$I_{ij}(x_i)$	An indicator function that relates a variable x_i to a task
M_i	The set of function nodes connected to x_i in a factor graph

N_j	The set of variable nodes connected to F_j in a factor graph
$\mathcal{M}_{\mathbf{x}_j \setminus x_i}$	Received messages by a function node F_j from all of its neighbouring variable nodes $\mathbf{x}_j$, other than x_i in a factor graph
$\mathbf{n}$	The total number of received messages by F_j (i.e. $\mathbf{n} = \lvert\mathcal{M}_{\mathbf{x}_j \setminus x_i}\rvert$)
m	The summation of the maximum values of each of the $\mathbf{n}$ messages of $\mathcal{M}_{\mathbf{x}_j \setminus x_i}$
$\mathcal{V}_i$	The array of sorted values of $\mathbf{s_i}$
p	The maximum value of $\mathcal{V}_i$
b	The summation of the corresponding values of p from the incoming $\mathbf{n}$ messages of F_j
t	The subtraction of b from m
$\mathbf{q}$	The minimum value of the proposed range within $\mathcal{V}_i$
$iNodes$	The variable and function nodes of F_G that are connected to minimum number of neighbours
$iNodes_{c_i}$	The variable and function nodes of cluster c_i that are connected to minimum number of neighbours
$pNodes$	The variable and function nodes of F_G that are permitted to generate and send a message at a certain time
$pNodes.pNeighbours$	The permitted neighbours of $pNodes$ in F_G
$A_m.iNodes$	The agents A_m that act on behalf of $iNodes$
$A'_m.pNodes$	The agents A'_m that act on behalf of $pNodes$
$iNodes.allNeighbours$	All the neighbours of $iNodes$
$generatedMessages$	All GDL messages generated at a certain time
T	The completion time of a GDL-based algorithm
T_{smp}	The completion time of a GDL-based algorithm following the SMP protocol
T_{pmp}	The completion time of a GDL-based algorithm following the PMP protocol
T_{intm}	The time required to complete the intermediate step of PMP
$\mathcal{N}_{\mathcal{F}}$	The number of function nodes in F_G
$\mathcal{N}_{\mathcal{C}}$	The number of clusters.
c_i	A single cluster.
$c_{largest}$	The largest cluster
$\mathcal{N}$	The maximum number of function nodes per cluster.
$firstFunction$	the cluster initiator function node.
A_c	The agent that holds the $firstFunction$.
$iNodes_{c_i}$	The variable and function nodes of cluster c_i that are connected to minimum number of neighbours
$min\_nFunction(F)$	Returns the functions which share variables with only one neighbouring function nodes.
$\mathcal{CH}$	The set of cluster heads

$\mathfrak{ch}_1$	A single cluster in $\mathcal{CH}$
$c_i.member()$	The set of member function nodes of cluster c_i
$adj(node)$	The neighbouring function node
$\mathcal{S}_\mathcal{N}$	The set clusters having only one neighbouring cluster each
$\forall Q(c_i)$	All variable-to-function messages within cluster c_i
$\forall R(c_i)$	All function-to-variable messages within cluster c_i
$Q_{ignEdge}(c_i)$	Edges of c_i that are ignored during the cluster formation process
$\mathcal{S}$	The set of split nodes in c_i
$\mathcal{S}_i$	The single split node in $\mathcal{S}$
$D_{F_j \to F_p}(x_i)$	Single computation within the intermediate step of PMP.
$dCount_{c_i}$	Number of neighbouring clusters of c_i
$\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$	The dependent acyclic graph for split node $\mathcal{S}_j$
$\mathcal{S}_j.values$	The recovered ignored values for split node $\mathcal{S}_j$
M_r	The message shared by the participating clusters of the first round of PMP.
T_{p_1}	Average time to compute a function-to-variable message within F_G
T_{p_2}	Average time to compute a variable-to-function message within F_G
T_{cm}	Average time to transmit a message between nodes of F_G
E1, E2, ..., E7	Seven different experimental settings to evaluate PMP as opposed to SMP
$\mathcal{W}_0, \mathcal{W}_1$	The regression coefficients.
$\mathcal{X}$	The independent or predictor variable.
$\mathcal{Y}$	The dependent or response variable.
$\mathcal{D}$	The set of training data for the regression analysis.
$\mathbb{A}$	A deployed message passing DCOP algorithm
$E_G(\mathbb{A}, P)$	Event-based dependency graph based on $\mathbb{A}$ and P
E_i	A single event in E_G
$\upsilon(E_i, E_j)$	The duration between the starting of E_i and the end of E_j.
E_i	A single event in E_G
$T(\mathbb{A}, P)$	The total completion time of $\mathbb{A}$ based on P
$deg(\eta_i)$	The number of connected neighbours of node η_i in G
k	The number of agents participating in the optimization process
N	The number of nodes in G
$uniformVal$	the ratio of N and k in G
λ	The set of control points in G
λ_i	A single control point in λ
$cap(A_j)$	The computational capability/processing power of agent A_j
l	Non-weighted path distance between two nodes of G

Acronyms

MAS	Multi-Agent Systems
DCSP	Distributed Constraint Satisfaction Problem
DCOP	Distributed Constraint Optimization Problem
GDL	Generalized Distributive Law
SMP	Standard Message Passing
PMP	Parallel Message Passing
BMS	Bounded Max-Sum
FMS	Fast Max-Sum
DFS	Depth First Search
BFS	Breadth First Search
NP	Nondeterministic Polynomial time
ADOPT	Asynchronous Distributed OPTimization
BnB	Branch and Bound
OptAPO	Optimal Asynchronous Partial Overlay
DPOP	Distributed Pseudotree Optimization Procedure
PC-DPOP	Partially Centralized Distributed Pseudotree Optimization Procedure
MB-DPOP	Memory Bounded Distributed Pseudotree Optimization Procedure
A-DPOP	Approximate Distributed Pseudotree Optimization Procedure
DJTG	Distributed Junction Tree Generator
DSA	Distributed Stochastic Algorithm
MGM	Maximal Gain Message

DBA	Distributed Breakout Algorithm
GHS	Gallager, Humblet and Spira
BnB-MS	Branch and Bound Max-Sum
BnB-FMS	Branch and Bound Fast Max-Sum
BFMS	Bounded Fast Max-Sum
MNA	Node-to-Agent Mapping
G-FBP	Generalized Fast Belief Propagation
GDP	Generic Domain Pruning
SECP	Smart Environment Configuration Problem
UAV	Unmanned Aerial Vehicle
GPU	Graphics Processing Unit
GP-GPU	General Purpose Graphics Processing Unit

Chapter 1

Introduction

Advances in technologies in recent years have triggered a significant growth in the use of intelligent physical devices (e.g. robots, wireless sensors, smart home appliances and unmanned aerial vehicles) in a broad range of application domains. These applications range from disaster response, climate research, military operations to industry/home and security arrangements (Jarvis et al., 2013). The varied nature of the applications reflects the diversity of the intelligent devices in terms of their capabilities, as well as their interactions with intelligent human operatives. In many of these situations, each of the individual intelligent entities, often referred to as an autonomous agent, needs to work with others as a team, rather than act on their own in order to achieve their individual goals, towards increasing the performance of the whole system (Jennings & Wooldridge, 1995). Thus, a number of cooperating agents with different capabilities form a multi-agent system (MAS) that can be deployed to accomplish a global objective. In such situations, an individual agent may need to take decisions that affect the performance of other agents within the MAS. Therefore, to obtain the best possible outcome from a system perspective, an effective coordination strategy that leads the agents to perform coordinated actions by considering their joint choices is crucial. To achieve this objective, significant attention has been given by the MAS research community to coordination techniques, in which the decision making process of each individual agent can be improved by taking into account the decisions of other agents within the system.

In the multi-agent systems literature, studies that address the coordination issue are based on different perspectives, including task/resource allocation, coalition formation and meeting scheduling. However, scalability remains an open issue for most of these. Specifically, scalability in MAS manifests itself as a performance problem caused by the inclusion of a large number of heterogeneous agents, their available resources and the tasks performed by the agents. Moreover, it is observed that coordination approaches often produce results in dealing with large settings, but take too much time in doing so (Farinelli et al., 2013; Lesser & Corkill, 2014; Fioretto et al., 2018). As a consequence, it is a common phenomenon that a coordination algorithm that works perfectly (or

reasonably) for a smaller setting does not cope with a larger setting. For example, often it is necessary to take a prompt decision when allocating rescue agents to save critical patients in a disaster response scenario or to obtain the highest profit from an e-commerce application (Ramchurn et al., 2010; Müller & Fischer, 2014). In such cases, and many others besides, the time required to obtain a solution from the coordination algorithm (the so-called completion time) should be minimized because the solution is likely to be outdated when it takes too long to produce.

In more detail, the completion time of a multi-agent coordination algorithm depends on two main elements: computation cost and communication cost (Lesser & Corkill, 2014). The former is the amount of computation that needs to be performed by the participating agents during the coordination process. The latter reflects the overall time elapsed during the interactions of the cooperating agents. Besides their impact on the completion time, there are other reasons for minimizing the computation and communication cost. For example, in many applications such as sensor networks or disaster response, agents possess a limited power back up, so it is undesirable to constantly keep the agents busy either for computing or for communicating (Vinyals et al., 2011). In any case, depending on the application, the computation cost can be significantly higher than the communication cost and vice versa. For example, the latter can be excessively expensive compared to the former in a disaster response scenario where communication facilities are disrupted. Considering the constraints that arise from the aforementioned background, it is important for a coordination approach to focus on reducing the cost of these two elements in order to minimize its completion time.

For over a decade, a significant amount of research has sought to address the coordination issue in the form of a constraint handling framework for different domains of MAS (e.g. disaster response, sensor networks and smart-homes). Approaches based on the constraint handling framework are suitable for large-scale settings as they generally produce results after a "one-shot" process (Farinelli et al., 2013; Leite et al., 2014). In the constraint handling framework, coordination problems are formulated as constraint networks that are often represented graphically. In particular, the agents are represented as nodes, and the constraints that arise between the agents depending on their joint choice of action are represented by edges (Dechter, 2003). Each constraint can be defined by a set of variables held by the corresponding agents related to the constraint, and denoted by a value (cost) or reward relation among the set of variables. In more detail, each agent holds one or more variables, each of which takes values from a finite and discrete domain. Each value in the domain represents a possible state of the agent. The agent is responsible for only setting the value of its own variable(s), but can communicate with other agents. Thus, agents are only aware of constraints that involve variables they hold. Such constraints are usually termed "local functions" and the sum of these local functions is the local utility of the agent. Here, two agents are considered neighbours

if there is at least one constraint that depends on variables held by each agent. Only neighbouring agents can directly communicate with each other.

The constraints that form the relationships among the agents in a graphical representation can either be hard or soft (Yokoo et al., 1998; Dechter, 2003). Hard constraints only consider relations that describe the accepted joint assignments of the variables. Unlike the hard constraints which returns a binary value for a particular assignment, a soft constraint is a real valued function that describes the cost or reward for each joint variable assignment. If a constraint network only considers hard constraints, the problem of finding a variable assignment that satisfies all the constraints is referred to as a Constraint Satisfaction Problem (CSP). In other words, a CSP can be described as a problem whose purpose is to find a combination of values for all variables, from their corresponding domains, such that all constraints are satisfied. Nevertheless, in MAS, computational resources are distributed across the network of interconnected agents, and this group of agents choose values for their variables in a distributed way. This is why Distributed Constraint Satisfaction Problems (DCSPs), which are the distributed version of CSPs, are a better fit for MAS (Yokoo et al., 1998).

Distributed Constraint Optimization Problems (DCOPs) only deal with soft constraints, and are problems in which agents choose values for their variables, such that the cost of a set of constraints over the variables is either minimized or maximized (Modi et al., 2005). In other words, a DCOP is similar to a DCSP except that the constraints return a real number instead of a boolean value, and the goal is to set every variable to a value from its domain to minimize the constraint violation. This is an assignment of all variables that optimizes a global function, which is an aggregation of the functions representing the constraints (i.e. local utilities). In a multi-agent system, it is usually more feasible to aim for the best possible solution rather than target a specific one. This is because the violation of even a single constraint among hundreds/thousands would mean the complete process is of no use. As a consequence, conceptualizing a multi-agent constraint handling problem setting as a DCOP is more appropriate as well as a natural approach than DCSP. This is therefore the focus of this thesis. Given this background, we formally define the key research requirements here.

- **R1:** The completion time of a DCOP solution approach employed for a MAS should be minimized while maintaining good or acceptable solution quality.
- **R2:** Minimize either or both the computation and communication cost of the participating agents, in order to speed-up the overall optimization process.
- **R3:** The employed DCOP algorithm should not be limited to one or few application specific setting(s), and should deal with a large (e.g. hundreds/thousands) number of cooperating agents, tasks and/or resources.

In light of these research requirements, we discuss possible solutions of DCOPs and the challenges that remain open for further research in Section 1.1. We then go on to outline our contributions to the solution of those challenges in Section 1.2. Finally, we detail the structure of the rest of this thesis in Section 1.3.

1.1 Distributed Constraint Optimization in Cooperative Multi-Agent Systems

Over the last decade, a number of algorithms have been developed to solve DCOPs (see Sections 2.3 and 2.4 for more details). They have been applied to many real world multi-agent applications in the form of task/resource allocation (Ramchurn et al., 2010; Zivan et al., 2014), meeting scheduling (Maheswaran et al., 2004b) and coalition formation (Cerquides et al., 2013). These algorithms can be broadly classified into exact and non-exact approaches. The former always finds an optimal solution, and can be further classified into fully decentralized and partially centralized approaches. In fully decentralized approaches (e.g. ADOPT (Modi et al., 2005), BnB ADOPT (Yeoh et al., 2008), DPOP (Petcu, 2005)), the agent has complete control over its variables and is aware of only local constraints. However, such approaches often require an excessive amount of communication when applied to complex problems (Petcu et al., 2007). On the other hand, centralizing parts of the problem can often reduce the effort required to find a globally optimal solution (e.g. OptAPO (Mailler & Lesser, 2004), PC-DPOP (Petcu et al., 2007)). In both cases, finding an optimal solution for a DCOP is an NP-hard problem that exhibits an exponentially increasing coordination overhead as the system grows (Modi et al., 2005). As a consequence, exact approaches are often impractical for application domains with larger constraint networks. On the contrary, non-exact algorithms sacrifice some solution quality for better scalability, and are thus more suitable to meet the research requirement **R3**. These algorithms are further categorized into local greedy and Generalized Distributive Law (GDL) based inference methods.

In general, local greedy algorithms, such as DSA (Fitzpatrick & Meertens, 2003) and MGM (Maheswaran et al., 2004a), begin with a random assignment of all the variables within a constraint network, and go on to perform a series of local moves that try to greedily optimize the objective function. They often perform well on small problems having constraints with lower arity. There has been some work on providing guarantees on the performance of such algorithms in larger DCOP settings (Kiekintveld et al., 2010; Pearce & Tambe, 2007; Bowring et al., 2008; Vinyals et al., 2010). However, for large and complex problems consisting of hundreds or thousands of nodes, this class of algorithms often comes up with a global solution far from the optimal (Farinelli et al., 2013; Leite et al., 2014; Zivan & Peled, 2012; Rogers et al., 2011). This is because agents do not explicitly communicate their utility for being in any particular state. Rather they only communicate their preferred state (i.e. the one that will maximise their own utility)

based on the current preferred state of their neighbours. Taken together, this particular class of algorithms is clearly not practically effective if we wish to meet the last part of our research requirement **R1**, along with **R3**.

Among the non-exact approaches, GDL-based inference algorithms, such as Max-Sum (Farinelli et al., 2008) and Bounded Max-Sum (BMS) (Rogers et al., 2011), have received particular attention. Agents in this class of algorithms calculate and propagate utilities (or costs) for each possible value assignment of their neighbouring agents' variables. Thus, the agents explicitly share the consequences of choosing non-preferred states with the preferred one during inference through a constraint graph. Eventually, this information helps these algorithms to achieve good solution quality for large and complex problems. Moreover, unlike many other DCOP solution approaches, GDL-based algorithms can directly handle n-ary constraints and more than one variable per agent (Cerquides et al., 2013; Farinelli et al., 2013). Consequently, these algorithms can easily be deployed to any DCOP setting without depending on an additional reformulation technique. Furthermore, they make efficient use of constrained computational and communication resources. This is achieved by following a message passing protocol in which the agents continuously exchange messages to compute an approximation of the impact that each of the agents' actions have on the global optimization function. This involves building a local objective function (detailed in Section 2.4.2). Once the function is built, each of the agents picks the value of a variable that maximizes the function. Taking into account all of these benefits, this class of algorithms are more suited to meet our key requirements, denoted as **R1**, **R2** and **R3**, than other DCOP solution approaches. Therefore, we specifically concentrate on the GDL-based non exact-approaches for this work.

Despite these aforementioned advantages, scalability remains a widely acknowledged challenge for GDL-based algorithms due to several reasons (Kim & Lesser, 2013; Ramchurn et al., 2010). Firstly, they perform repetitive maximization operations for each constraint function to select the locally best configuration of the associated variables, given the local utility function and a set of incoming messages. To be precise, a constraint function that depends on n variables having domains composed of d values each, will need to perform d^n computations for a maximization operation. As the system scales up, the complexity of this step grows exponentially, eventually making this class of algorithms too expensive in terms of computational cost. Several attempts have been made to reduce the cost of the maximization operation. However, they are either tailored to a specific domain, and as such, are not applicable to general DCOP settings, or they experience lack of consistency (see Section 2.5.2 for more details).

Secondly, it is worth mentioning that previous attempts at speeding up[1] GDL-based algorithms have mainly focused on reducing the overall cost of the maximization operator.

[1]They attempt to speed-up existing approaches in order to improve their scalability, and we are specifically influenced by this insight.

However, they overlook an important concern that all the GDL-based algorithms follow a Standard Message Passing (SMP) protocol to exchange messages among the nodes of a constraint graph that represents the corresponding DCOP. In the SMP protocol, *a message is sent from a node v on an edge e to its neighbouring node w if and only if all the messages are received at v on edges other than e, summarized for the node associated with e* (Aji & McEliece, 2000; Kschischang et al., 2001). This means that a node in the constraint graph is not permitted to send a message to its neighbouring node until it receives messages from all its other neighbours. Here, for w to be able to generate and send messages to all its other neighbours, it depends on the message from v. To be exact, w cannot compute and transmit messages to its neighbours other than v until it has received all essential messages, including the message from v. This dependency, which is common for all the nodes, produces increasing amounts of average waiting time for agents as the graphical representation of a DCOP becomes larger. As a consequence, the time required to obtain the solution from the DCOP algorithm (i.e. the completion time) increases. Besides the synchronous SMP protocol, there is an asynchronous version of message passing where nodes are initialized randomly, and outgoing messages can be updated at any time and in any sequence (Farinelli et al., 2008). Nevertheless, the completion time for both the cases are proportional to the diameter of the constraint graph, while the asynchronous version never outperforms SMP in terms of the completion time (see Section 2.5.3 for details). In either case, therefore, the ensuing increase of delay, produced by currently used message passing protocols, with the corresponding constraint graph remains a key issue for scalability.

Finally, a number of representations, such as Depth First Search (DFS) trees (Modi et al., 2005; Petcu, 2005), junction trees (Aji & McEliece, 2000; Stefanovitch et al., 2011) and factor graphs (Kschischang et al., 2001; Farinelli et al., 2008), have been used by different DCOP algorithms as the constraint graph (see Section 2.2 for detail discussion). In all of these representations, nodes (i.e. variables and/or factors depending on the graphical representation) are being held[2] by the agents participating in the optimization process. In a conventional DCOP model, it is assumed that this mapping of nodes of a DCOP graphical representation to the participating agents is part of the model description. Although this assumption is often reasonable, there are many applications where there is some flexibility in making this assignment (e.g. assignment of UAVs to target to survey). While it is possible to arbitrarily choose a mapping and run any off-the-shelf DCOP algorithm to solve the problem, choosing a good mapping is important as it can have a significant impact on an algorithm's completion time (as we shall expound in Section 5.1). However, choosing an optimal mapping may be prohibitively time consuming as this is an NP-hard problem (Rust et al., 2016), and there is currently no generic heuristic algorithm that efficiently maps the nodes to the agents (see Section 2.5.4 for details).

[2]The agents act (i.e. generate and transmit messages) on behalf of the nodes they hold.

Summarily, in light of the above discussion, the following are the key research challenges we will address in this thesis.

- **C1:** Reduce the computation cost of the maximization operation of GDL-based algorithms while maintaining the same solution quality and preserving generic applicability.
- **C2:** Speed up the message passing process of GDL-based algorithms while maintaining the same solution quality and preserving generic applicability.
- **C3:** Speed up GDL-based algorithms via an efficient Node-to-Agent mapping heuristic that does not affect the solution quality or applicability.

1.2 Research Contributions

In the context of the challenges outlined in the previous section, the contributions of the work presented in this thesis are as follows. Note that these contributions operate independently of one another; however, they can be used in the same deployed algorithm to accelerate the respective phases.

1. We propose a **G**eneric **D**omain **P**runing technique, that we call $\mathcal{GDP}$, in order to prune the search space during the execution of the maximization operation of GDL-based algorithms (Chapter 3). $\mathcal{GDP}$ is applicable to all DCOP settings, and we theoretically prove that the pruned search space obtained by our approach does not alter the solution quality of such algorithms (**C1**). Finally, we empirically evaluate the performance of $\mathcal{GDP}$, and observe a significant reduction of search space (i.e. $33\% - 81\%$) by using this technique. More importantly, we illustrate that the relative performance gain of our approach gets better with an increase in the parameters the maximization operator acts on. This work has been accepted for publication in a conference (Khan et al., 2018a).

2. We introduce a new generic message passing protocol for GDL-based algorithms, that we call Parallel Message Passing ($\mathcal{PMP}$). Specifically, we utilize the advantages of partial centralization without being affected by its major shortcomings (see Chapter 4). To be precise, $\mathcal{PMP}$ combines clustering with domain pruning, as well as the use of a regression method to determine the appropriate number of clusters for a given scenario while maintaining the same solution quality. In so doing, $\mathcal{PMP}$ efficiently distributes the computational and communication overhead to the agents, such that their computational power is concurrently exploited, and their average waiting time before sending messages to solve DCOPs using the GDL framework is reduced (**C2**). We empirically evaluate the performance of our method in a number of settings and find that it brings down the completion time by around $37-85\%$ ($1.6-6.5$ times faster) for $100-900$ nodes, and by around $47-91\%$

$(1.9 - 11$ times faster) for $3000 - 10000$ nodes compared to the current state-of-the-art. A part of this study is presented in a workshop (Khan et al., 2016), and the complete work is published in a journal (Khan et al., 2018b).

3. We formulate the problem of node-to-agent mapping as an optimization problem, where the goal is to find an assignment that minimizes the completion time of the GDL-based algorithms that operate on this mapping. Then, we propose a new time-efficient heuristic to determine a near-optimal **M**apping of **N**odes to the participating **A**gents, that we call $\mathcal{MNA}$ (see Chapter 5). $\mathcal{MNA}$ is a pre-processing step that works prior to executing the optimization process of a GDL-based DCOP algorithm. Specifically, we propose two versions (i.e. centralized and decentralized) of $\mathcal{MNA}$ that can be used considering the application at hand. As a pre-processing step, our approach does not alter any internal process of such DCOP algorithm; therefore, it does not have any impact on its solution quality (**C3**). Additionally, the decentralized version of $\mathcal{MNA}$ specifically caters for scenarios where the graphical representation experiences change(s) during runtime. We empirically evaluate the performance of $\mathcal{MNA}$ in terms of completion time, and show that it performs at a level of around $90\% - 100\%$ of the optimal mapping, which is computationally infeasible to obtain in practice. Our results also show a speed-up of $16\% - 40\%$ compared to the current state-of-the-art, meaning a GDL-based algorithm can perform $1.2 - 1.7$ times faster when using node-to-agent mappings generated by $\mathcal{MNA}$. This work has been accepted for publication in a conference (Khan et al., 2018c).

Overall, the work in this thesis has led to the publication of the following papers.

1. Khan, M. M., Tran-Thanh, L., & Jennings, N. R. (2018a). A generic domain pruning technique for gdl-based dcop algorithms in cooperative multi-agent systems. In *Proceedings of the 17th International Conference on Autonomous Agents and Multi-Agent Systems* (accepted as a full paper) (pp. $1 - 9$).: IFAAMAS.

2. Khan, M. M., Tran-Thanh, L., Ramchurn, S. D., & Jennings, N. R. (2018b). Speeding up gdl-based message passing algorithms for large-scale dcops. *The Computer Journal* (pp. $1 - 28$).: Oxford University Press.

3. Khan, M. M., Tran-Thanh, L., Yeoh, W., & Jennings, N. R. (2018c). A near-optimal node-to-agent mapping heuristic for gdl-based dcop algorithms in multi-agent systems. In *Proceedings of the 17th International Conference on Autonomous Agents and Multi-Agent Systems* (accepted as a full paper) (pp. $1-9$).: IFAAMAS.

4. Khan, M. M., Ramchurn, S. D., Tran-Thanh, L., & Jennings, N. R. (2016). Speeding up gdl-based message passing algorithms for large-scale dcops. In *International Workshop on Optimization in Multi-Agent Systems (OPTMAS'2016)* (pp. $1-15$).

1.3 Thesis Outline

The remaining part of this thesis is structured as follows.

- In Chapter 2, we provide a review of related literature with a focus on the research areas and challenges mentioned in Section 1.1.
- Chapter 3 presents and evaluates $\mathcal{GDP}$, which reduces the computation cost of the potentially expensive maximization operation. That is the first of the three ways we speed-up GDL-based DCOP algorithms in cooperative multi-agent systems.
- In Chapter 4, we address the second challenge by speeding up the message passing process of GDL-based algorithms. We develop a new message passing protocol, and discuss the technical details of how it can minimize the completion time while maintaining the same solution quality compared to the current state-of-the-art. We then report comparative empirical results. At the end, we demonstrate the details and the performance of applying the linear regression model on top of the proposed protocol.
- In Chapter 5, we speed-up GDL-based algorithms via an effective node-to-agent mapping (i.e. the third way). In so doing, we present a near-optimal node-to-agent mapping heuristic in the form of its centralized and decentralized versions, and evaluate its performance empirically against the current state-of-the-art.
- Chapter 6 draws conclusions to the work explained in Chapters 3, 4 and 5, and highlights areas of future work.

Chapter 2

Literature Review

In this chapter, we survey the literature of multi-agent systems (MAS) to explore different approaches that have been applied to solve Distributed Constraint Optimization Problems (DCOPs). In Section 2.1, we formally define the generic DCOP framework. Then, we discuss three main forms of constraint graphs that have been used to represent DCOPs graphically (Section 2.2). Section 2.3 presents various exact approaches for solving DCOPs, and discusses the shortcomings of these approaches in dealing with large and complex DCOPs. To be precise, we begin with different search-based exact algorithms in Section 2.3.1. We then detail the Generalized Distributed Law (GDL) framework, and discuss a number of prominent exact algorithms that have been inspired by this framework (Sections 2.3.2 - 2.3.3). Subsequently, Section 2.4 discusses two major classes of non-exact DCOP solution approaches, namely local greedy algorithms and GDL-based algorithms. We end this section by highlighting the significance of GDL-based non-exact approaches with regards to the scalability issue, and discussing how speeding up such algorithms can further improve their applicability in terms of scalability. Next, in Section 2.5, we identify a number of expensive phases of GDL-based algorithms, and investigate current approaches that have been proposed for speeding them up. Finally, we provide a summary of our findings in Section 2.6.

2.1 Distributed Constraint Optimization Problems

Distributed Constraint Optimization Problems (DCOPs) are a widely studied framework for coordinating interactions in cooperative multi-agent systems. DCOPs have gained such popularity because of their ability to optimize a global objective function that can be described as the aggregation of a number of distributed constraint cost functions. More specifically, this framework has been frequently used in multi-agent applications where participating agents must come to some form of agreement. Such an agreement leads them to do some actions jointly to obtain the best possible solution for the complete

system (e.g. coalition formation, meeting scheduling, task allocation). Typically, when multiple, perhaps all, solutions are valid some are usually preferred to others. One of the key aspects of DCOPs for multi-agent settings relies on the fact that each agent normally negotiates locally with just a subset of other agents (i.e. neighbours) which have direct influence on its actions. In more detail, Cerquides et al. (2013); Farinelli et al. (2013); Leite et al. (2014) and Fioretto et al. (2018) summarize the major approaches and the applications of DCOPs from different perspectives of MAS. In general, a DCOP can be formally defined as follows.

Definition 2.1. *(DCOP).* A DCOP can be defined by a tuple $\langle$A, X, D, F, $\delta\rangle$, where

- A is a set of agents $\{A_1, A_2, \dots A_k\}$.
- X is a set of finite and discrete variables $\{x_0, x_1, \dots, x_m\}$, which are being held by the set of agents A.
- D is a set of domains $\{D_0, D_1, \dots, D_m\}$, where each $D_i \in D$ is a finite set containing the values to which its associated variable x_i may be assigned.
- F is a set of constraints $\{F_1, F_2, \dots, F_{\mathcal{L}}\}$, where each $F_i \in F$ is a function dependent on a subset of variables $\mathbf{x}_i \in X$ defining the relationship among the variables in $\mathbf{x}_i$. Thus, function $F_i(\mathbf{x}_i)$ denotes the value for each possible assignment of the variables in $\mathbf{x}_i$ and represents the joint pay-off that the corresponding agents achieve. Note that this setting is not limited to pairwise (binary) constraints, and the functions may depend on any number of variables.
- δ is a function $\delta : \eta \rightarrow A$ that represents the mapping of variables (and functions), jointly denoted by η, to their associated agents. Each $\eta_i \in \eta$ is held by a single agent. However, each agent can hold several η_i.

Notably, the dependencies (i.e. constraints) between the variables and functions generate a constraint graph G. Within this model, the objective of a DCOP algorithm is to have each agent assign values to its associated variables from their corresponding domains in order to either maximize or minimize the aggregated global objective function, which eventually produces the value of all the variables, X^* (Equation 2.1).

$$X^* = \arg\max_X \sum_{i=1}^{\mathcal{L}} F_i(\mathbf{x}_i) \;\; OR \;\; X^* = \arg\min_X \sum_{i=1}^{\mathcal{L}} F_i(\mathbf{x}_i) \tag{2.1}$$

Although a constraint graph G is a standard, as well as a natural way to visualize a DCOP instance, most of the DCOP algorithms require certain structure and ordering among the components (i.e. nodes, agents) of G. To reconcile the differences in choice, each of the algorithms makes use of a pre-processing step which operates before executing the original optimization process (Modi et al., 2005; Farinelli et al., 2008). This

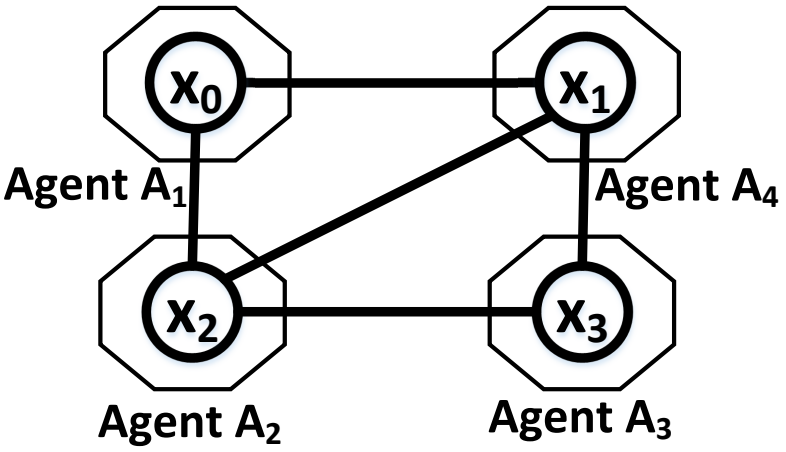


Figure 2.1: A sample constraint graph representation of a DCOP, with four variable nodes $\{x_0, x_1, x_2, x_3\}$ being held by four agents $\{A_1, A_2, A_3, A_4\}$. In the figure, variables are denoted by circles and agents are octagons.

phenomenon, particularly, leads to different graphical representations corresponding to G. Note that the selection of a graphical representation for a DCOP plays a fundamental role, both from an agent coordination perspective and from an algorithmic perspective (Fioretto et al., 2018). In the following section, we are going to discuss predominant representations that have been adapted for various DCOP algorithms.

2.2 Graphical Representations of DCOPs

There are three common graphical representations upon which the development of different DCOP algorithms have been based. In other words, a DCOP algorithm operates directly on one of the following three graphical representations: Depth First Search (DFS) trees (Modi et al., 2005; Petcu, 2005), junction trees (Aji & McEliece, 2000; Stefanovitch et al., 2011) or factor graphs (Kschischang et al., 2001; Farinelli et al., 2008). In the remainder of this section, we are going to discuss each of them in detail.

2.2.1 Depth First Search Tree

A number of DCOP algorithms have been developed based on the fact that the nodes of a constraint graph G must be arranged in a depth first search (DFS) order. Formally, a DFS arrangement of a constraint graph G is a rooted tree with the same nodes and edges as G, and the property that adjacent nodes from the original graph fall in the same branch of the tree (Petcu, 2007). Figure 2.1 illustrates an exemplary constraint graph, and a sample corresponding DFS-tree arrangement is depicted in Figure 2.2. In both of these figures, variables $\{x_0, x_1, x_2, x_3\}$ are denoted by circles, while agents $\{A_1, A_2, A_3, A_4\}$ are denoted by octagons. In a DFS-tree, direct parent-child relationship between agents are depicted via tree edges (solid black lines). For instance, agent A_1

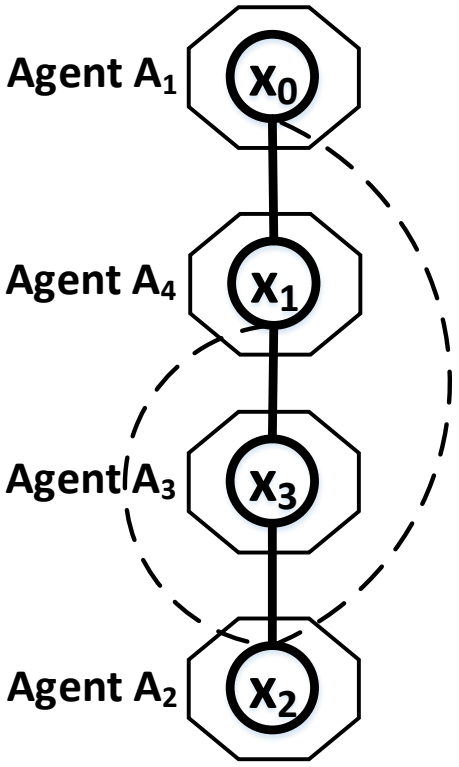


Figure 2.2: Constraint graph of Figure 2.1 arranged as a DFS-tree.

is the parent of agent A_4 in Figure 2.2. In addition to the tree edges, there exists back edges (dashed black lines) which symbolize pseudo parent/child relationships between the agents of a DFS-tree. Specifically, these are the edges of G that are not identified as the tree edges during the DFS traversal process. In the example, agent A_2 is the pseudo-child of agent A_4.

Notably, the property of DFS-tree, which ensures the adjacent nodes from the original graph G fall in the same branch of the tree, is significant in the perspective of the distributive nature of DCOPs. This is because agents in different branches of the tree do not share any constraints, so that they can search for solutions independently of each other. Despite this benefit, a DFS-tree experiences several shortcomings when applied to a DCOP. First, each agent can hold only one variable of the DFS-tree representation of a DCOP, while a general DCOP definition allows multiple nodes per agent. Second, it cannot directly deal with n-ary constraints (i.e. functions involving more than two variables). In more detail, a DCOP with binary constraints (i.e. relations) is typically represented by a specific type of constraint graph, called primal-constraint graph, whose nodes stand for variables and edges stand for binary constraints (Dechter, 2003). DFS-trees are considered as a possible arrangement for this type of constraint graphs. To overcome these shortcomings, a DCOP algorithm designed over DFS-tree has to rely on one or more additional reformulation technique(s), which requires changes in the algorithm design (Burke & Brown, 2006; Yokoo, 2001; Bowring et al., 2006; Pecora et al., 2006). This incurs an added complexity from an algorithmic perspective.

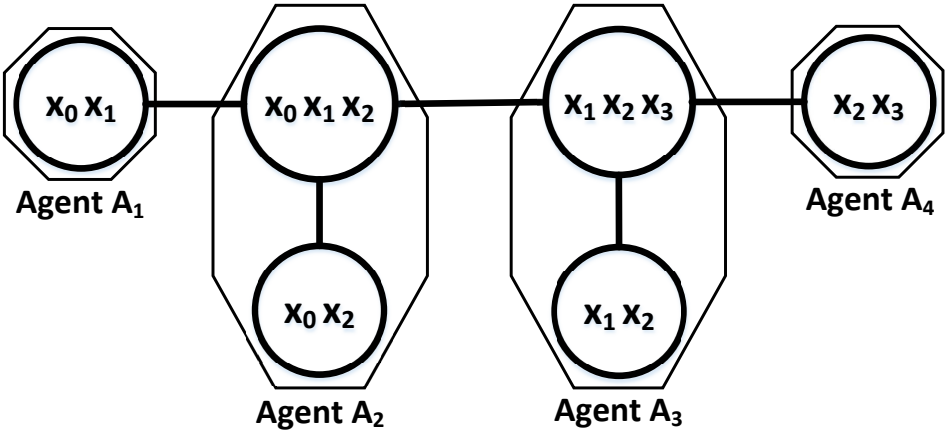


Figure 2.3: Constraint graph of Figure 2.1 arranged as a junction tree. Here, the intersection of any two nodes remain as a subset of a node in the path of those two nodes. For example, $\{x_2\}$ remains in the path of the nodes $\{x_0, x_2\}$ and $\{x_1, x_2\}$. Moreover, if the tree is projected onto any variable, such as x_1, it yields a tree as well.

2.2.2 Junction Tree

Another way to represent a DCOP is through a junction tree, a phenomenon that originated from the fact that DCOPs can also be represented by a different type of constraint graph (namely dual-constraint graph) whose nodes stand for constraints and whose edges link constraints that share some variable in their domains (Aji & McEliece, 2000; Dechter, 2003). Specifically, a junction tree is a clique graph that satisfies following properties (the so-called junction tree properties). Firstly, a clique is a maximal and complete cluster of nodes (i.e. subset of variables), such that if a node x_i has a link to all of the nodes in clique U, node x_i belongs to clique U. Secondly, if a variable x_i appears in both clique U and V, then x_i should remain in all cliques on the path between U and V. Thirdly, a junction tree is a tree which yields another tree when projected onto each individual variable. Figure 2.3 illustrates a sample junction tree transformed from the exemplar constraint graph of Figure 2.1. By considering the above mentioned properties, it is relatively trivial to determine whether a junction tree exists or not for a given constraint graph. Moreover, unlike a DFS-tree, a junction tree can directly handle n-ary constraints and more than one variable per agent (Stefanovitch et al., 2011). As a consequence, it can easily be deployed to any DCOP setting without depending on an additional reformulation technique. However, finding the optimal junction tree in the sense of a minimal size of the largest clique is itself an NP-hard problem (Aji & McEliece, 2000). Hence, DCOP algorithms developed over junction trees are required to use one of the distributed heuristics proposed in the MAS literature. Moreover, it is often difficult to visualize the factorization property of large and complex DCOPs from the usual depictions of graphs as utilized in junction trees (Wainwright & Jordan, 2008). This is why, despite having a number of advantages over DFS-tree, junction tree does not gain much popularity among the DCOP community.

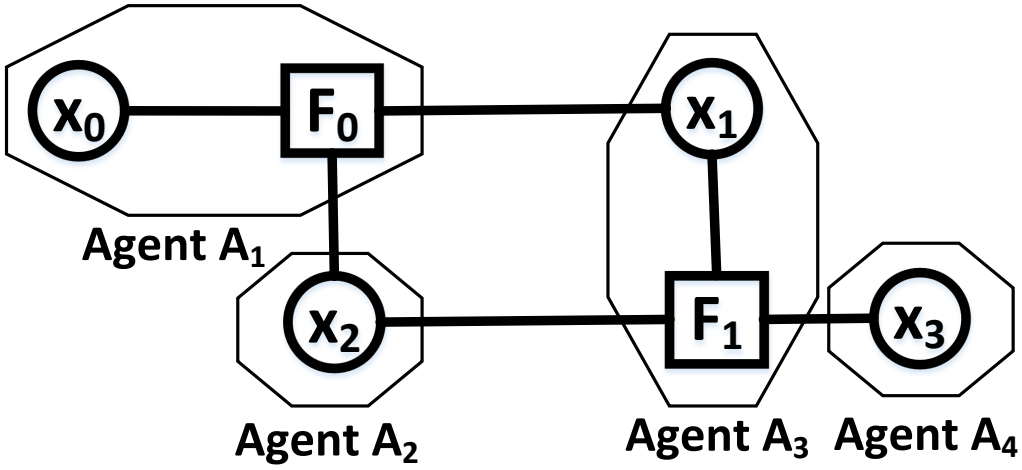


Figure 2.4: A sample factor graph representation of the constraint graph of Figure 2.1. In the figure, variables are denoted by circles, factors are squares and agents are octagons.

2.2.3 Factor Graph

The formalism of factor graphs provides an alternative graphical representation, one which emphasizes the factorization of the variable distribution, unlike the previous two representations. Specifically, factor graphs are a straightforward generalization of Tanner graphs (Kschischang et al., 2001). The Tanner graph was originally used to depict constraints imposed for error recovery using parity checking. The basic problem of error-control coding is that of transmitting a message (a sequence of bits) through a noisy channel in such a way that a receiver can recover the original message despite the noise.

Definition 2.2. *(Factor Graph).* A factor graph is a bipartite graph that expresses the structure of the factorization. It has a variable node for each variable, a factor/ function node for each local function and an edge that connects a variable node to a factor/function node if and only if that variable is an argument of that function.

Similar to the Tanner graph, a factor graph (Definition 2.2) is a bipartite graph, meaning it has two types of nodes, and each node only has neighbours of the opposite type. This graphical model represents the factorization of a function. Similar to junction tree, it can directly handle n-ary constraints and more than one variable per agent (Farinelli et al., 2013; Fioretto et al., 2018). Moreover, it experiences an additional axiomatic benefit, that is because of the bipartite nature of factor graph, it can easily distinguish the relationship between the constraints and the variables of a DCOP. Initially, Kschischang et al. (2001) proposed the use of the factor graph instead of the junction tree for the belief propagation or probability propagation algorithm. Afterwards, a number of DCOP algorithms that operates on the factor graph representation of a DCOP have been developed (see Section 2.4.2 for details). Figure 2.4 shows an instance of such a problem represented effectively by a factor graph.

Building on these graphical representations, numerous studies have attempted to propose algorithms to solve DCOPs in the context of MAS. Some of these algorithms are generic while others focused on specific application domains. They can be broadly categorized as exact and non-exact algorithms. In the following two sections, we briefly discuss some of the most prominent exact and non-exact DCOP solution approaches.

2.3 Exact DCOP Algorithms

Exact algorithms always find a solution that optimizes the global objective function for a DCOP instance. Some of these algorithms are search-based and the rest are inference-based, the later typically inspired by the Generalized Distributive Law (GDL) framework. Both of these classes of DCOP algorithms can be further categorized as fully decentralized and partially centralized approaches. In the remainder of this section, we describe some key exact algorithms.

2.3.1 Search-Based Exact Algorithms

ADOPT (Asynchronous Distributed OPTimization) is a search-based fully decentralized exact algorithm that allows agents to asynchronously change the values of the variables they hold (Modi et al., 2005). It uses the best-first strategy as a search technique where each agent always assigns the best possible value to its variable(s) based on the local information. ADOPT has a preprocessing step, where all the agents participating in the optimization process are arranged in a DFS-tree (see Section 2.2.1). After forming the DFS-tree from the corresponding constraint graph, ADOPT starts its message propagation along the edges of the tree. Specifically, three different types of messages, *Value*, *Cost* and *Threshold*, are exchanged during this process. At the beginning, each agent chooses a random value for the variable it holds and initializes its lower bound (zero) and upper bound (infinity). These bounds are iteratively refined over time. The whole process terminates when the agents find a value for which their bounds are equal. To this end, each agent sends its descendants the value with the smallest lower bound through the *Value* message, then waits for *Cost* messages to come back from the children and adds them to the lower bound of its current value. *Threshold* messages are sent from parents to children to update the agent's backtrack thresholds. ADOPT uses thresholds to store the lower bound previously computed for its current context, such that it can reconstruct the partial solution more efficiently. Thus, the backtrack thresholds are useful when a previously visited context is re-visited.

Yeoh et al. (2008) propose another search based exact algorithm named BnB-ADOPT, which inherits most of the messages and data structures of ADOPT. However, BnB-ADOPT employs the depth-first branch-and-bound search strategy unlike ADOPT,

which uses the best-first search approach. This particular difference changes the behaviour of the agents in terms of sharing their values. In ADOPT, each agent chooses the value that minimizes its lower bound. On the other hand, BnB-ADOPT changes the value of a variable for a particular context only when it is able to find out that the optimal solution for that value is not better than the current best solution. In addition, also unlike ADOPT, each agent in BnB-ADOPT uses *thresholds* to store the cost of the current best solution for all contexts and uses them to change its values more efficiently. Thus, BnB-ADOPT consistently reduces computation cost by using a depth-first search strategy in previously formed DFS tree representation of the DCOP.

A number of other attempts have been made to improve the performance of ADOPT and BnB-ADOPT. In particular, Ali et al. (2005) introduce the use of pre-processing techniques for guiding ADOPT search, and prove that it eventually increases the performance consistently. Moreover, researchers observed that many messages sent by BnB-ADOPT are redundant, and an extension of BnB-ADOPT, BnB-ADOPT$^+$, without most of the redundant messages is proposed (Gutierrez & Meseguer, 2010). As a result of exchanging fewer messages, it performs better than the original BnB-ADOPT algorithm in terms of communication cost. Notably, the search-based exact algorithms either use best-first or depth-first search strategy in order to solve DCOPs. Both of them have their own shortcomings. For instance, the former often has to deal with a large number of solution reconstructions, while the later is unable to promptly prune sub-optimal branch. In order to remedy their weaknesses, Chen et al. (2017a) propose the use of a combination of those two search strategies based on agents' positions in a DFS-tree.

Unlike the algorithms discussed above, the Optimal Asynchronous Partial Overlay (OptAPO) is an exact DCOP algorithm that maintains partial decentralization (Mailler & Lesser, 2006). It was originally proposed to solve DCSPs, but can easily be adopted for DCOPs. OptAPO discovers the hard portions of the problem through trial and error and centralizes these sub-problems into mediating agent(s). These mediating agents posses higher processing power than other agents within the system. The authors show that the message complexity of OptAPO is significantly smaller than that of ADOPT. However, in order to guarantee that an optimal solution has been found, one or more agents may end up centralizing the entire problem, depending on the difficulty of the problem and the tightness of the interdependence between the variables. As a consequence, it is impossible to predict where and what portion of the problem will be eventually centralized, or how much computation the mediator(s) have to resolve. Nonetheless, it is possible that several mediators needlessly duplicate effort by solving overlapping problems.

In general, the search-based exact algorithms require polynomial memory in terms of their message size. However, their main downside is that they produce a very large number of small messages due to their asynchronous nature, resulting in large communication overheads (Petcu, 2007). In contrast, inference-based exact algorithms mainly rely on synchronous message passing protocols. As a consequence, they produce a linear

number of messages. Given the fact that the GDL framework plays an important role in developing inference-based algorithms, we first discuss the framework in the subsection that follows.

2.3.2 The Generalized Distributive Law (GDL) Framework

The distributive law is one of the most frequently used properties in mathematics. It comes from the fact that multiplying a number is the same as multiplying its addends by the number, then adding the products. In general, it exploits the ability of one operation to "distribute" over another operation contained inside a set of parenthesis. Through the distributive law, it is possible to compute equations such as $(pq+pr)$ where $\{p, q, r\} \in \mathbb{R}(real\ number)$ in a faster way. This is because the equivalent equation $p(q + r)$ obtained by applying the distributive law involves two arithmetic operations as opposed to three operations required by the former. The following example shows how significant the computational difference is. Suppose, $f_1(x, y, w)$ and $f_2(x, z)$ are two real valued functions, where x, y, z and w variables take values from the domain D of size n. The global function $F(w, x)$ in the left side of Equation 2.2 can be obtained by marginalizing out the variables y and z from $f_1(x, y, w)f_2(x, z)$. Thus it takes $2n^4$ arithmetic operations. On the other hand, because of the distributive law, the same global function can be computed by the expense of total $n^3 + n^2 + n^2$ or $n^3 + 2n^2$ equations only, which is much less than $2n^4$ (Equation 2.3).

$$F(w, x) = \sum_{y,z \in D} f_1(x, y, w) f_2(x, z) \tag{2.2}$$

$$F(w, x) = (\sum_{y \in D} f_1(x, y, w))(\sum_{z \in D} f_2(x, z)) \tag{2.3}$$

At this point, the computational gain that can be obtained by using the distributive law is obvious from the above example. Although the computational savings can be dramatic in more complicated cases, it is not trivial to recognize the impact through raw mathematical expressions. This is why it is necessary to have a general framework to handle such problems coming from a wide range of applications. The Generalized Distributive Law (GDL) accomplishes this objective by passing messages in a communication network which is often represented graphically using a junction tree or a factor graph (Aji & McEliece, 2000; Kschischang et al., 2001).

Given this background, it is clear that GDL can greatly reduce the number of additions and multiplications required in a certain class of computational problems. Specifically, much of the power of the GDL lies in the fact that it can be applied to situations in which the notions of addition and multiplication are themselves generalized. The appropriate framework for this generalization is the commutative semiring (Definition 2.3).

Definition 2.3. *(Commutative Semiring).* A commutative semiring is a set $\mathcal{K}$, along with two binary operations called "+" and "$\cdot$", which satisfy the following three axioms.

- The operation "+" is associative and commutative, and there is an additive identity element called "0" such that $k + 0 = k$ where $\forall k \in \mathcal{K}$.
- The operation "$\cdot$" is also associative and commutative, and there is an multiplicative identity element called "1" such that $k \cdot 1 = k$ where $\forall k \in \mathcal{K}$.
- The distributive law holds, that is $(a \cdot b) + (a \cdot c) = a \cdot (b + c)$ where $\{a, b, c\} \in \mathcal{K}$.

From the definition of commutative semiring, we know that 0 and 1 are the identity elements for the operators "+" and ".", respectively. The generalization we highlighted can be realized from the following instance: if those two operators are replaced by the symbols max and + having identity elements $-\infty$ and 0 respectively, it forms a commutative semiring named Maximum-Summation or Max-Sum. Equation 2.4 depicts an example of distributed law for the semiring Max-Sum.

$$\max(a + b, a + c) = a + \max(b, c) \tag{2.4}$$

In a nutshell, GDL is a unifying framework for performing inference in a graphical model, and this framework operates on commutative semirings through message propagation. We obtain different message passing algorithms, such as Sum-Product, Max-Product, Max-Sum or Min-Sum, depending on the specific semiring used.

2.3.3 GDL-Based Exact Algorithms

Distributed Pseudotree Optimization Procedure (DPOP) and Action-GDL are two key exact inference DCOP algorithms inspired by the GDL framework. In the remainder of this section, we detail both of them along with their variants.

2.3.3.1 DPOP and Its Variants

DPOP was the first exact inference-based DCOP algorithm, proposed by Petcu (2005). Specifically, it is a utility propagation method originated using the concept of GDL. Since GDL-based approaches are correct only for acrylic/tree-shaped constraint graphs (expounded in Section 2.4.2), they propagate an additional type of message (the so called *value* propagation) in order to provide optimal solutions for cyclic graphical representations. Similar to ADOPT and BnB-ADOPT, DPOP works on a DFS-tree representation of a DCOP. Nevertheless, unlike those asynchronous search-based approaches, DPOP propagates messages in a synchronous manner. In more detail, the algorithm operates

in two phases over a DFS-tree. First, agents compute the optimal utility by performing summations and maximizations (i.e. the GDL semiring operators) flowing from the leafs to the root of the tree. Then, the optimal assignment is obtained using the *value* propagation procedure down the tree. To be precise, DPOP performs the following three phases in sequence.

- *DFS* traversal: An agent is selected as root, preferably by a leader election process. From the root, a distributed DFS traversal of the constraint graph is started. At the end, each agent labels its neighbours as parents, pseudo-parents, children or pseudo-children, and edges are identified as tree or back edges. The DFS-tree serves as a communication structure for the following phases.
- *Util* propagation: the agents, starting from the leaves, send *Util* messages to their parents. Each of these messages contains an aggregated cost function that is computed by adding received *Util* messages from its children with its own cost functions as well as its parent and pseudo-parents', and then projecting out its own variables by optimizing over them.
- *Value* propagation: The agent at the root starts this phase. Then, each of the agents determines their optimal value using the cost function computed in *Util* propagation phase and the *Value* message(s) received from their parent(s). Afterwards, they inform their children using *Value* messages.

In the literature, there are several extensions of DPOP that have been proposed to address various trade-offs in the approach. Specifically, the cycle-cut set idea for addressing the trade-off between the number of messages used and the amount of memory that each message requires is exploited by MB-DPOP (Petcu & Faltings, 2007). Another method, A-DPOP, addresses the trade-off between message size and solution quality. It reduces message size by optimally computing only a part of the messages and approximating the rest with upper and lower bounds (Petcu, 2007). Moreover, Petcu et al. (2007) introduces PC-DPOP, a partial decentralization technique based on DPOP, in order to take the benefits experienced by OptAPO (the partially decentralized search-based approach). However, unlike OptAPO, it offers prior exact predictions about communication, computation and memory requirements. Additionally, PC-DPOP provides better control than OptAPO over what parts of the problem are centralized, though this control cannot be guaranteed to be optimal. Furthermore, Chen et al. (2017b) recently claim that practically constructed DFS-trees often come to be chain-like and that phenomenon greatly impairs the performances of DPOP algorithms. To overcome this, they propose the use of a Breadth First Search (BFS) traversal instead of DFS, in order to form the corresponding graphical representation of a DCOP. However, they observe that BFS-tree often generates a very large utility message with improper allocations. Consequently, their approach requires an additional step to deal with this.

2.3.3.2 Action GDL

Action-GDL is another inference-based exact algorithm which is more explicitly associated with GDL framework than DPOP (Vinyals et al., 2009). Action-GDL itself operates exactly similar to DPOP. More specifically, both Action-GDL and DPOP have two similar phases (i.e. *Util* and *Value* phases) for their message propagation, and use exactly the same two GDL operators, summation and maximization, in their respective *Util* phases. However, unlike DPOP, Action-GDL operates over a junction tree instead of a DFS-tree. To be exact, they propose to use a Distributed Junction Tree Generator (DJTG) algorithm in order to capture the distribution of relations in DCOPs. As aforementioned, unlike the DFS-tree, a junction tree can directly represent n-ary constraints and allows more than one node per agent, thus it is more suitable to represent complex problems. As a result, Action-GDL experiences some axiomatic benefit over DPOP in terms of the amount of computation, communication and parallelism of the algorithm solving cost.

In any case, although inference-based algorithms require a linear number of messages compared to search approaches, the problem is that the messages (i.e. *Util* messages) exchanged by the agents in DPOP/Action GDL may be exponentially large. This reflects an exponential increase of computation cost and memory requirements. On the contrary, we observe that the search algorithms can be infeasible to deploy due to their communication cost (i.e. a large number of small messages exchanged). This is obvious because finding an optimal solution for a DCOP is an NP-hard problem. Therefore, multi-agent DCOPs, where a large number of agents, tasks and resources are involved, are too expensive (computation and/or communication) to solve using the exact algorithms. This phenomenon significantly limits their applicability and provides opportunity for non-exact approaches.

2.4 Non-Exact DCOP Algorithms

As discussed earlier, all the exact algorithms calculate the globally optimal solution. However, achieving optimality demands an exponentially increasing coordination overhead. Such exponential relationships are unsuitable for a multi-agent system where agents are constrained in computational capacity, bandwidth and memory resources. Under these circumstances, non-exact algorithms are appropriate, as they trade some solution quality for scalability, such that they have been used more in practice (Fitzpatrick & Meertens, 2003; Maheswaran et al., 2004a; Farinelli et al., 2008). In the remainder of this section, we discuss various non-exact algorithms that are broadly classified into local greedy search and GDL-based inference approaches.

2.4.1 Local Greedy Non-Exact Algorithms

In general, local greedy algorithms begin with a random assignment of all the variables within a constraint graph, and go on to perform a series of local moves that try to greedily optimize the objective function. Normally, changing the value of a small set of variables is involved in a local move so that the difference between the value of the objective function with the new assignment and the previous value is maximized. This is defined as the gain. The search terminates when the procedure obtains a local maximum, implying that there is no local move available which offers a positive gain. In other words, agents in a local search algorithm perform and evaluate the local moves in parallel, and inform their neighbours about the assignment after each move. Local greedy search is a popular non-exact optimization technique, as it requires very little memory and computation, and can obtain good solutions in many settings. However, the parallel execution without coordination can produce poor system performance, since these agents can act based on outdated knowledge about the choices of other agents. To address this, a number of efforts have been made that introduce different key algorithms of this type for solving DCOPs, such as Distributed Stochastic Algorithm (DSA) (Fitzpatrick & Meertens, 2003), Distributed Breakout Algorithm (DBA) (Yokoo & Hirayama, 1996; Hirayama & Yokoo, 2005) and Maximum Gain Message (MGM) (Maheswaran et al., 2004a).

Specifically, DSA can be used for both synchronous and asynchronous contexts. In the initialization phase of the former version, each of the agents chooses a random value for the variable it holds and enters in an infinite loop. Within the loop, an agent chooses an activation probability $p_i \in [0, 1]$ and generates a random number $r_i < p_i$. Then, the agent will choose a value for its variable x_i only if r_i is less than p_i. Otherwise it will not change the current value of x_i. Afterwards, the agent sends the updated information to all neighbours and they, in turn, update information accordingly. On the other hand, the asynchronous version of DSA can be used when the rate of the variable change is low. Notably, the optimization step considers only the current values of neighbours, and the communication step involves a message to communicate just the new value of the agent's variable. Hence, there is no exponential increase in computation and communication. Finally, the algorithm is anticipated to increase the solution quality after each execution step. Thus the longer an agent waits before acting, the better the solution will be and importantly the solution is always available. This any-time property of DSA is suitable for many practical applications. However, coupled with the lack of theoretical guarantees, the solution quality is strongly dependent on the activation probability p_i. Unfortunately, there is no proactive way of computing this value from an analysis of the problem instance, which is the main drawback of DSA.

On the other hand, in DBA, agents initiate the optimization process by assigning values equal to one to all constraints. Subsequently, each of the participating agents randomly

chooses a value for the variable it holds, then it propagates the selected value and obtains the values chosen by its neighbouring agents. Once any of the agents receives all the values chosen by the neighbours, it computes the assignment that results in a global cost reduction by considering the weights of the violated constraints. At the end, the agents share this cost reduction among their neighbours. When the cost reduction is larger than zero, the corresponding agent assigns the value computed for this reduction. Otherwise, it adds one to all the weights of all the violated constraints. The overall search process ends when there is no violation of constraints. Since the main objective of the DCSP model is to ensure that there is no constraint violation (see Chapter 1), DBA is usually more appropriate for DCSPs than DCOPs.

The MGM algorithm is typically considered as the DCOP version of DBA, wherein the difference lies in the fact that MGM does not alter the cost of the constraints to avoid a local maximum. Similar to DBA, MGM is executed synchronously where all the agents perform their local computations in each round of the algorithm. Each agent starts the process by choosing an arbitrary value for its variable. Afterwards, the agents propagate the selected values and obtain the values chosen by neighbouring agents. After receiving the values chosen by the neighbours, each agent selects a value that results in better unilateral gain, and propagates the gain among its neighbouring agents. The agent assigns the selected value if the gain computed locally is greater than the maximum gain of all neighbouring agents. To be precise, MGM performs a distributed hill climbing; however, the algorithm avoids local maximum when computing the maximum improvement from its neighbour prior to assigning the selected value. The rounds in MGM are repeated until a ending condition is satisfied. Since the agents compute unilateral actions, MGM does not guarantee convergence to the optimal solution. Unlike DSA, which requires a single cycle (i.e. only the value propagation), MGM requires two cycles in each round. For the first cycle, the agents propagate their assigned values. For the second cycle, the agents propagate the obtained gains by unilateral changes.

Overall, these local greedy approaches often perform well on small problems having constraints with lower arity. There has been some work on providing guarantees on the performance of such algorithms in larger DCOP settings (Kiekintveld et al., 2010; Pearce & Tambe, 2007; Bowring et al., 2008; Vinyals et al., 2010). However, for large and complex problems consisting of hundreds or thousands of nodes, this class of algorithms often comes up with a global solution far from the optimal (Farinelli et al., 2013; Leite et al., 2014; Zivan & Peled, 2012; Rogers et al., 2011). This is because agents do not explicitly communicate their utility for being in any particular state. Rather they only communicate their preferred state (i.e. the one that will maximise their own utility) based on the current preferred state of their neighbours.

2.4.2 GDL-Based Non-Exact Algorithms

Among the non-exact approaches, GDL-based algorithms are receiving increasing attention. They gain notable computational savings by using the distributive law (see Section 2.3.2). Moreover, unlike greedy approaches, agents in a GDL-based algorithm explicitly share the consequences of choosing non-preferred states with the preferred one during inference through the graphical representation of a DCOP (Farinelli et al., 2008; Ramchurn et al., 2010; Leite et al., 2014). Thus, agents can obtain global utilities for each possible value assignment. In other words, in contrast to the greedy local search algorithms, agents do not propagate assignments. Instead, they calculate utilities for each possible value assignment of their neighbouring agents' variables. Eventually, this information helps this class of algorithms to achieve good solution quality for large and complex problems. Hence, they perform well in practical applications.

Although the GDL framework has been utilized before in exact approaches, non-exact GDL-based algorithms essentially inherit most of its benefits by preserving the identical means of computation. Moreover, unlike exact inference algorithms, they do not introduce any additional phase of message passing – for example, an additional *value* propagation phase is used by DPOP and Action-GDL. Furthermore, this class of algorithms specifically operates on the factor graph representation of a DCOP, which overcomes several of the shortcomings of the junction tree representation while inheriting the benefits it experiences over DFS-tree (see Section 2.2.3). In the remainder of this section, we discuss two key GDL-based non-exact algorithms in greater detail.

2.4.2.1 The Max-Sum Algorithm

As discussed in Section 2.3.2, the "generalization" property of GDL framework allows a certain application to choose its preferred commutative semiring operators that are required to perform inference in a graphical model. In this way, we obtain different message passing algorithms such as Sum-Product, Max-Sum or Max-Product, depending on the specific semiring used. These algorithms are widely used to perform inference in different domains. Initially, the GDL framework gained much popularity with the Max-Product algorithm in the domain of probabilistic belief propagation (Kschischang et al., 2001). However, from the perspective of MAS, the global utility of a set of agents can be expressed more efficiently by "summing" the local utilities, instead of computing their "product". This is why the semiring operator *sum* used by the semiring Max-Sum is more suited in this case than the operator *product* used by Max-Product. Building on this insight, Farinelli et al. (2008) proposed the use of the Max-Sum semiring for solving DCOPs in MAS.

Definition 2.4. *(Standard Message Passing (SMP) protocol).* Within a graphical representation of a DCOP, a message is sent from a node v on an edge e to its neighbouring

node w if and only if all the messages are received at v on edges other than e, summarized for the node associated with e.

Similar to the Max-Product algorithm, Max-Sum operates on a factor graph representation of a DCOP. Specifically, it follows a standard message passing protocol (so called SMP protocol – Definition 2.4) to exchange messages among the nodes of the factor graph (Kschischang et al., 2001; Farinelli et al., 2008). According to SMP, a node (variable or function) in the factor graph is not permitted to send a message to its neighbouring node until it receives messages from all its other neighbours. For example, in the factor graph of Figure 2.4, the function node F_1 can send a message to the variable node x_3 only after receiving the messages from its other neighbouring variable nodes, x_1 and x_2. In other words, F_2 has to wait for the messages to come from x_1 and x_2 before sending a message to x_3, according to the SMP protocol.

$$Q_{x_i \to F_j}(x_i) = \sum_{F_k \in M_i \setminus F_j} R_{F_k \to x_i}(x_i) \tag{2.5}$$

$$R_{F_j \to x_i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} [F_j(\mathbf{x}_j) + \sum_{x_k \in N_j \setminus x_i} Q_{x_k \to F_j}(x_k)] \tag{2.6}$$

$$Z_i(x_i) = \sum_{F_j \in M_i} R_{F_j \to x_i}(x_i) \tag{2.7}$$

Now, the algorithm uses Equations 2.5 and 2.6 for the message passing. Specifically, the variable and the function nodes of a factor graph continuously exchange messages (variable x_i to function F_j (Equation 2.5) and function F_j to variable x_i (Equation 2.6)) to compute an approximation of the impact that each of the agents' actions have on the global objective function, by building a local objective function $Z_i(x_i)$. Once the function is built (Equation 2.7), each agent picks the value of a variable that maximizes the function by finding $\arg\max_{x_i}(Z_i(x_i))$. In Equations 2.5−2.7, M_i stands for the set of functions connected to x_i and N_j represents the set of variables connected to F_j.

According to the SMP protocol, the iterative message passing process of Max-Sum terminates after each variable receives messages from all its connected neighbours. Now, there is an asynchronous version of message passing where nodes are initialized randomly, and outgoing messages can be updated at any time and in any sequence (Farinelli et al., 2008). Thus, the asynchronous protocol minimizes the waiting time of the agents, but there is no guarantee about how consistent their local views (i.e. the local objective function) are. In other words, agents can take decisions from an inconsistent view and they may need to revise their action. Therefore, unlike SMP, even in an acyclic factor graph, this asynchronous version does not guarantee convergence after a fixed number

of message exchanges. Thus, it experiences more communication and computational cost as redundant messages are generated and sent, regardless of the structure of the graph. As a consequence, the asynchronous property limits the inference algorithms' axiomatic benefit of communicating a limited number of messages compared to search-based algorithms. Significantly, even in the asynchronous version, the expected result for a particular node can be achieved only when all the received messages for the node are computed by following the regulation of the SMP protocol. Based on this observation, Elidan et al. (2006) introduced an asynchronous propagation algorithm that schedules messages in an informed way. Moreover, Peri & Meisels (2013) demonstrated that the impact of inconsistent views is worse than the waiting time of the agents regarding the total completion time, due to the effort required to revise an action in the asynchronous protocol. Thus, the completion time for both the cases are proportional to the diameter of the factor graph, and the asynchronous version never outperforms SMP in terms of the completion time (Kschischang et al., 2001; Peri & Meisels, 2013; Leite et al., 2014).

In any case, due to several reasons, Max-Sum is often considered as one of the prominent non-exact algorithms. Firstly, in Max-Sum, agents only need to have a local perspective of the complete problem and they do not rely on greedy moves. Secondly, the size of the messages exchanged in Max-Sum depends only on the domain size of the variables, thus either they are small or increase linearly with variables' domain size. Whereas, the message size of exact inference algorithms (e.g. DPOP and Action-GDL) increases exponentially with either or both variables' domain size and constraints' arity. Then, the number of messages exchanged varies linearly with the number of nodes within the system. Thirdly, in Max-Sum, agents do not need to hold the connectivity graph in memory and update it as new nodes are joined or leave because it does not need the agents to organize themselves into a DFS-tree, unlike most other complete DCOP algorithms (see Section 2.3). Fourthly, this algorithm can directly handle n-ary constraints and more variables per agent. Finally, Max-Sum provides optimal solution when applied to an acyclic factor graph. However, it is not guaranteed to converge when applied to general factor graphs which typically contain loops/cycles. We are going to discuss an extension of Max-Sum that remedied this particular concern in the subsection that follows.

2.4.2.2 The Bounded Max-Sum Algorithm

Building on the fact that Max-Sum is neither guaranteed to converge nor provide optimal solutions for cyclic factor graphs, Rogers et al. (2011) introduced Bounded Max-Sum (BMS) that provides a bounded approximate solution by removing cycles in the factor graph. This is achieved by ignoring the dependencies between functions and variables which have the least impact on the solution quality. In more detail, BMS is able to bound the quality of the solutions found by removing a subset of edges from a cyclic

factor graph to make it acyclic, and by running Max-Sum to solve the acyclic problem. Specifically, the goal is to compute a variable assignment $\tilde{X}$ in the transformed acyclic factor graph, such that $V^* \leq \rho\tilde{V}$, where the approximate solution $\tilde{V} = \sum_{i=1}^{\mathcal{L}} F_i(\tilde{\mathbf{x}}_i)$ and $V^* = \sum_{i=1}^{\mathcal{L}} F_i(\mathbf{x}_i^*)$ is the solution based on the optimal variable assignment. In this context, ρ represents the approximation ratio. BMS prunes the edges that have the least impact on solution quality in order to keep this approximation ratio as small as possible. The maximum impact of an edge between x_j and F_i is defined as its weight w_{ij}. Thus, if variable x_j is ignored when maximising a function F_i, the distance between the approximate and the optimal solution will be at most w_{ij} (Equation 2.8).

$$w_{ij} = \max_{\mathbf{x}_i \setminus x_j} [\max_{x_j} F_i(\mathbf{x}_i) - \min_{x_j} F_i(\mathbf{x}_i)] \tag{2.8}$$

Once all the weights are computed, BMS uses a modified version of a distributed minimum spanning tree algorithm developed by Gallager, Humblet and Spira (GHS), to generate the maximum spanning tree (Gallager et al., 1983). GHS is reasonable in terms of communication cost ($O(N \log N + E)$) and has a running time of $O(N \log N)$, where N is the number of nodes in the factor graph and E denotes the number of edges. The newly obtained acyclic factor graph is then used in the second phase, where the Max-Sum algorithm is used to compute the optimal variable assignment to the modified problem (Equation 2.9).

$$\tilde{V} = \arg\max_{X} \sum_{i} \min_{\mathbf{x}_i^c} F_i(\mathbf{x}_i) \tag{2.9}$$

Here, $\mathbf{x}_i^c$ is the set of variables that were eliminated from the scope of a function F_i, corresponding to the edges that were pruned from the factor graph. Now, the approximation ratio ρ can be obtained by Equation 2.10.

$$\rho = 1 + (\tilde{V} + W - \tilde{V})/\tilde{V} \tag{2.10}$$

Here, W is the sum of the weights of the pruned edges. Thus, an upper bound on the optimal solution can be computed by Equation 2.11.

$$\tilde{V} + W \geq V^* \tag{2.11}$$

Notably, the BMS algorithm has been enhanced in terms of solution quality by decomposing the utility functions and using tighter upper and lower bounds (Rollon & Larrosa, 2014). Besides, a number of other efforts have been made to improve the applicability and scalability of Max-Sum and BMS algorithms. We explore these in the following section.

2.5 Speeding Up GDL-Based DCOP Algorithms

Based on the discussion undertaken in the previous section, it becomes apparent that GDL-based non-exact algorithms, such as Max-Sum and BMS, are good candidates for dealing with DCOPs in practical multi-agent systems. Nevertheless, scalability remains a key challenge for them due to a number of potentially expensive phases. In the remainder of this chapter, we examine different approaches that have been proposed to reduce those phases' costs, and finally summarize our key findings.

2.5.1 Constraint Graph Formation

As discussed in Section 2.2, the internal optimization process of a DCOP algorithm is usually followed by a pre-processing step, which is used to generate a corresponding constraint graphical representation such as DFS-tree, junction tree or factor graph. It has also been observed that none of these representations are unique for a corresponding DCOP. Instead, it is common to have many variants of a particular graphical representation where each variant corresponds to the same problem (Aji & McEliece, 2000; Modi et al., 2005). That being the case, it is important for a DCOP algorithm to choose the best of the variants, as it would ensure a lessened completion time for the deployed algorithm (Petcu, 2007; Stefanovitch et al., 2011). Moreover, the attribute that defines the optimality is predicated on the choice of the graphical representation. For example, a minimal size of the tree width is preferred in the context of DFS-tree, whereas an optimal junction tree should have the largest clique of minimal size. In any case, finding an optimal representation that corresponds to a DCOP is itself an NP-hard problem. Therefore, DCOP solution approaches usually rely on a distributed heuristic (e.g. Awerbuch (1985); Stefanovitch et al. (2011)) in order to form a suitable variant of a representation corresponding to a particular DCOP.

2.5.2 Maximization Operation

As mentioned in Section 2.4.2, both the Max-Sum and BMS algorithms use Equations 2.5 and 2.6 for their message passing, and they can be directly applied to the corresponding factor graph representation of a DCOP. Specifically, Equation 2.6 stands for a message that is computed by each of the factor nodes (i.e. constraint functions) distinctly for all of its neighbouring variable nodes. Each of these factor-to-variable messages comprises a maximization operation that is used to obtain the locally best configuration of the associated variables, given the local utility function and a set of incoming messages. To be exact, a constraint function that is associated with n variables having domains composed of d values each, will need to perform d^n computations to complete a maximization operation. The computational complexity of this operator grows exponentially as the system scales up. In other words, because of the potentially large

Algorithm 1: Algorithm for computing BnB-MS domain pruning message from function F_j to variable x_i.

1 *Compute* $F_j(x_i)^{ub} \leq \min_{\mathbf{x}_j \setminus x_i} F(x_i, \mathbf{x}_j \setminus x_i)$

2 *Compute* $F_j(x_i)^{lb} \geq \max_{\mathbf{x}_j \setminus x_i} F(x_i, \mathbf{x}_j \setminus x_i)$

3 *Send* $< F_j(x_i)^{ub}, F_j(x_i)^{lb} >$ *to* x_i

parameter domain size and constraint functions with high arity, the maximization operator of the factor-to-variable message is one of the main reasons GDL-based algorithms can be computationally expensive.

Against this backdrop, Stranders et al. (2009) propose an extension of Max-Sum, named as Branch and Bound Max-Sum (BnB-MS), to reduce the computation cost of this step (Algorithm 1). Specifically, they perform a branch and bound search with constraint functions that ensure the upper and lower bound can be evaluated with a subset of variable values (lines $1-2$ of Algorithm 1), and then send to the receiving variable x_i (line 3). Here, $\mathbf{x}_j$ stands for those subset of variables the sending function F_j associated to. However, they introduce BnB-MS to be applied for the coordination of mobile sensors. More importantly, the bounding function they propose to compute the upper and lower bounds is solely focused on this specific application domain. Hence, this method is not directly applicable to other DCOP settings.

Another application specific approach that has been proposed to speed-up the maximization operation is named as the Fast Max-Sum (FMS) algorithm (Ramchurn et al., 2010). In particular, FMS reduces the domain size of variables associated with constraint functions for task allocation domains where agents' action choices are strictly divided into working on a task or not. In more detail, unlike standard Max-Sum, where each function node must enumerate all valid states and choose the one with the maximum utility, FMS restricts the domain of each variable to only two states for each connected function node. One of the states represents the fact that an agent A_i is assigned to a specific task t_j and the other state indicates that task t_j is not allocated to the agent A_i. The function $I_{ij}(x_i) \in \{0, 1\}$ acts as an indicator which returns 1 if $x_i = t_j$, and 0 otherwise. The revised variable-to-function and function-to-variable messages that reflect the above mentioned changes are shown in Equations 2.12 and 2.13, respectively. Where M_i denotes the set of indices of functions connected to x_i and N_j denotes the set of indices of variables connected to F_j in the factor graph. Notice that this is different from the Max-Sum which would have searched assignments of the variable which do not improve the utility of the factor in any way. Thus, FMS prunes the space that would have originally been searched by Max-Sum without losing any information. When a variable x_i receives all the messages from its neighbours, it computes the local objective function $Z_i(x_i)$ using Equation 2.14. Then, the variable can choose which value it takes as $\arg\max_{x_i}(Z_i(x_i))$ as before. Thus, it reduces the domain size (state) of each variable

from d to 2, compared to a standard implementation of Max-Sum. Hence, the computational complexity of a factor with n variables of domain size d in FMS is $\mathcal{O}(2^n)$, in contrast to $\mathcal{O}(d^n)$ required by applying Max-Sum to the same environment. Similar to Max-Sum, FMS is not guaranteed to converge in cyclic factor graphs. To overcome this challenge, Macarthur (2011) combines BMS with FMS to provide a bounded approximate solution in this particular formulation of a task allocation problem. Moreover, Macarthur et al. (2011) applies BnB-MS on top of the FMS algorithm to deal with the task allocation problem in mobile sensor application domain.

$$Q_{x_i \to F_j}(I_{ij}(x_i)) = \begin{cases} \sum_{k \in M_i \setminus F_j} R_{F_k \to x_i}(0) & // \quad if\ I_{ij}(x_i) = 1 \\ \max\limits_{b \neq j} [R_{F_b \to x_i}(1) + \sum_{k \in M_i \setminus F_b, F_j} R_{F_k \to x_i}(0)] & // \quad otherwise \end{cases} \tag{2.12}$$

$$R_{F_j \to x_i}(I_{ij}(x_i)) = \max_{x_j \setminus i} [F_j(\mathbf{x}_j) + \sum_{k \in N_j \setminus x_i} Q_{x_k \to F_j}(I_{kj}(x_k))] \tag{2.13}$$

$$Z_i(x_i) = R_{F_j \to x_i}(1) + \sum_{k \in N_j \setminus x_i} R_{F_k \to x_i}(0) \tag{2.14}$$

Tarlow et al. (2010) have shown that the computation associated to belief propagation algorithms can be reduced to polynomial time for some specific types of factors, known as Tractable Higher Order Potentials(THOPs). Nevertheless, they admit that not all DCOP settings can be represented using THOPs, and a notable limiting feature of THOPs is that they can only be defined or formulated over binary constraints. In particular, Pujol-Gonzalez et al. (2015) utilize THOPs to run Max-Sum in polynomial time for the domain of task allocation in RoboCup Rescue. Meanwhile, Kim & Lesser (2013) propose a more general approach, namely Generalized Fast Belief Propagation (G-FBP), to speed up the maximization operation of Max-Sum/BMS. Specifically, G-FBP uses two partially sorted lists in order to find the maximum of the summation as in Equation 2.6. The *value list* and the *message list* are the two partially sorted lists used here. The *value list* corresponds to a partially sorted version of the constraint function F_j given the specific value of a single variable. On the other hand, the *message list* represents a partially sorted list corresponding to the sum of incoming messages to a function node. Notably, they select and sort only the top $cd^{(n-1)/2}$ items of both lists where d is the domain size, n is the number of associated variables and c is a constant. The main intuition behind such a select-then-sort operation is that for the maximization operation, only the top $cd^{(n-1)/2}$ items will be accessed most of the time; unsorted entries are not accessed in most cases. Nevertheless, Kim and Lesser admit that they cannot guarantee in advance whether the presumption is true or false, and in the latter case G-FBP incurs a significant penalty in terms of the computational cost.

2.5.3 Message Passing Process

In reviewing the literature, we have found that previous attempts at speeding up GDL-based non-exact algorithms have mainly focused on reducing the overall cost of the maximization operator. However, they overlook an important concern that all such algorithms follow a message passing protocol, the so-called SMP (Definition 2.4), to exchange messages among the nodes of the corresponding factor graph representation of a DCOP. It is apparent from the discussion of Definition 2.4 that a node (variable or function) of a factor graph has to rely on other message(s) to receive before generating its own message(s). This dependency, which is common for all the nodes, increases the average waiting time for agents as the graphical representation of a DCOP becomes larger. As a consequence, the time required to obtain the solution from the DCOP algorithm (i.e. the completion time) increases. Although there is an asynchronous alternative of this protocol, this issue of taking too long remains identical because of the fact that an algorithm converges under similar conditions for both versions of the protocol (see Section 2.4.2.1 for details). Thus far, no studies have been performed to speed-up the message passing process, despite the significance of this issue. Nevertheless, it is worth mentioning that there have been few approaches that change the order of the propagation of the messages with the intention to improve the convergence property (i.e. solution quality) in the cyclic graphical representations (Vinyals et al., 2010; Elidan et al., 2006; Zivan & Peled, 2012).

2.5.4 Node-to-Agent Mapping

As mentioned in Section 2.2, DCOPs are formulated as constraint networks that are often represented graphically using one of the following representations: junction trees, factor graphs or DFS-trees. In all of these representations, nodes (i.e. variables and/or functions depending on the graphical representation) are being held by the agents participating in the optimization process, and the agents act (i.e. generate and transmit messages) on behalf of the nodes they hold.

The conventional DCOP model assumes that the mapping of nodes to the participating agents is part of the model description. In other words, the nodes that each agent holds is given as an input. This assumption is reasonable in many applications where there are obvious and intuitive mappings – for example, in a smart home scheduling problem (Fioretto et al., 2017), agents correspond to the different smart homes, and variables (i.e. nodes) correspond to the different smart devices within each home. In this case, the agent controls all the variables that map to the devices in its home. However, in other applications, there may be more flexibility in the mapping of nodes to agents. For example, imagine an application where a team of unmanned aerial vehicles (UAVs) need to coordinate with each other to effectively survey an area. In this application,

agents correspond to UAVs and variables correspond to the different zones in the area to be surveyed. The domain for each variable may correspond to the different types of sensors to be used and/or the different times to survey the zone. Since a UAV can survey any zone, there are multiple possible assignments of zones to UAVs. That is, there are multiple possible mappings of variables (i.e. nodes) to agents.

Although it is possible to arbitrarily choose a mapping and run any off-the-shelf DCOP algorithm to solve the problem, choosing a good mapping is important as it can have a significant impact on an algorithm's completion time (as we shall see in Section 5.1). However, choosing an optimal mapping may be prohibitively time consuming as this is an NP-hard problem (Rust et al., 2016). To date, this particular issue has received scant attention in the research literature. Notably, Rust et al. (2016) introduce a simple heuristic of node-to-agent mapping, based on constraints that arise from a specific problem formulation of a smart-home application, called Smart Environment Configuration Problem (SECP). Therefore, this method cannot be applied to other DCOP settings. In any case, the method does not aim to speed-up the overall DCOP algorithm, rather concentrating on establishing the roles of each smart-home devices.

2.6 Summary

Based on our survey of the literature, we outline the key shortcomings of existing works and highlight elements that we intend to build upon with a view to address the challenges mentioned in the previous chapter. Even though exact algorithms such as ADOPT, BnB-ADOPT, DPOP, Action-GDL, PC-DPOP and OptAPO always provide an optimal solution for a DCOP, they incur exponential coordination overhead. Consequently, such algorithms are unsuitable for settings involving a large number of agents, tasks and/or resources. Nevertheless, it is worth mentioning that centralizing parts of a DCOP can often reduce the effort required to find a globally optimal solution. To that end, approaches such as PC-DPOP and OptAPO have been proposed to take advantage of this phenomenon. In any case, non-exact algorithms have gained prominence as potential realistic alternatives. They sacrifice some solution quality for scalability and have, therefore, been used more in practice. Typically, they are based on local information exchange and can render acceptable solution quality while being efficient in terms of the computation and communication cost. Among the non-exact algorithms, DSA, DBA and MGM are local greedy search-based. They do not necessitate extensive computation/memory and provide good solutions for some applications. However, local best choices picked by a greedy method often translate into a global solution far from the optimal one for relatively large and complex multi-agent systems.

On the other hand, GDL-based inference non-exact methods, such as Max-Sum and BMS, do not rely on greedy moves. They try to maximize the global optimization function expressed as the sum of local functions by exchanging messages in a factor graph. The messages here are generated using the GDL framework that has an axiomatic tendency of computational savings. Moreover, they also provide some additional advantages by using a factor graph as the graphical representation of a DCOP. Furthermore, they ensure an optimal solution for acyclic factor graphs and bounded approximate solution for cyclic ones. Under such circumstances, this class of algorithms has great potential of being effective in large multi-agent settings.

Despite these advantages, scalability continues to pose a challenge for GDL-based algorithms. In reviewing the literature, we specifically find the presence of a number of potentially expensive phases within this class of algorithms, which can make them infeasible with regard to either or both computation and communication cost. Over the past few years, a number of attempts have been made to speed-up these phases, and in effect, improve the scalability and applicability of such algorithms. The majority of these attempts mainly focus on reducing the computation cost of the maximization operation. In particular, FMS and BFMS reduce the domain size of variables associated with constraint functions (for the purpose of task allocation domains) where the agents' action choices are strictly divided into working on a task or not. However, this method is completely application dependent, because it can only be applied to a specific problem formulation of the task allocation domain. Moreover, BnB-MS performs a branch and bound search with constraint functions that ensure the upper and lower bound can be evaluated with a subset of variable values. However, the bounding function they propose to achieve this is solely focused on coordinating mobile sensors. Hence, it is not directly applicable to general DCOP settings. In contrast, G-FBP is a more general approach proposed to reduce the cost of the maximization operator. In this approach, they select and sort the top $cd^{\frac{n-1}{2}}$ values of the search space, presuming that the maximum value can be found from these ranges. Here, c is a constant. Nevertheless, no guarantee can be made in advance about whether the presumption is true or false; in the latter case, G-FBP incurs a hefty penalty in terms of the computational cost. When taken together, it is obvious that the existing approaches of addressing this particular issue are unable to address our key research challenge **C1** (see Chapter 1). To address this issue, we develop a domain pruning technique to speed up the maximization operation in Chapter 3. This is something that can function regardless of any application dependency.

Another potentially expensive phase of GDL-based algorithms is the message passing protocol that ensures the manner in which nodes of a graphical representation exchange messages among themselves. On the basis of the agents' average waiting time, the currently used standard message passing protocol is eventually seen to increase the duration of the overall optimization process. Notably, no studies have been conducted thus far

to speed-up this process. In order to bridge the gap, and ultimately address the research challenge outlined as **C2**, we introduce a generic message passing protocol for GDL-based algorithms that can be used in place of existing protocol in Chapter 4.

Finally, the node-to-agent mapping, which explicitly defines the responsibility of the cooperating agents as well as their relationships with the nodes in the optimization process, signifies one of the key phases of GDL-based algorithms that can potentially increase the algorithms' completion time. While very few attempts have been made targeting this particular issue, they are generally tailored for specific application domain. This insight inspires the final contribution of our thesis where we speed-up the overall optimization process through a time-efficient node-to-agent mapping heuristic, and as such, successfully address our final research challenge **C3** (Chapter 5).

Chapter 3

Speeding Up the Maximization Operation

As discussed in Chapter 2, scalability becomes a challenge for Generalized Distributive Law (GDL) based message passing algorithms, such as Max-Sum and Bounded Max-Sum (BMS), when they have to deal with constraint functions with high arity or variables with a large domain size. In either case, the ensuing exponential growth of search space can make the maximization operation of such algorithms computationally infeasible in practice. Moreover, the existing studies that have been attempted to reduce the cost of this particular phase of GDL-based DCOP algorithms generally focus on a specific application domain and/or experience lack of consistency in their performance. Under such circumstances, it is important to speed-up the maximization operation to enable the effective implementation of these algorithms on larger and more complex problems, which is outlined as our first research challenge **C1**. This can be attained by reducing its computation cost whilst maintaining consistent performance and general applicability.

Building on this insight, this chapter proposes a **G**eneric **D**omain **P**runing technique, that we call $\mathcal{GDP}$, that is applicable to all DCOP settings. To be exact, $\mathcal{GDP}$ operates as a part of the maximization operator, proveably without affecting its solution quality (see *Lemma* 1 of Section 3.3). In other words, we improve the computational efficiency of non-exact GDL-based algorithms by reducing the search space over which the maximization operation is computed. More importantly, we show the relative performance gain of $\mathcal{GDP}$ gets better with an increase in the domain size of the variables and the arity of the corresponding constraint function. The remainder of this chapter is structured as follows. We describe the problem in greater detail in the section that follows (Section 3.1). In Section 3.2, we discuss the complete process of $\mathcal{GDP}$ with a worked example. We then discuss theoretical results in Section 3.3. Subsequently, in Section 3.4, we present the empirical results of our method compared to the current state-of-the-art. Finally, we conclude this chapter with the overall summary in Section 3.5.

3.1 Problem Description

As mentioned in Section 2.1, a DCOP can be defined by a tuple $\langle X, D, F, A, \delta \rangle$, where X is a set of discrete variables $\{x_0, x_1, \ldots, x_m\}$ and $D = \{D_0, D_1, \ldots, D_m\}$ is a set of discrete and finite variable domains. Each variable x_i can take value from the states of the domain D_i. F is a set of constraint functions $\{F_1, F_2, \ldots, F_{\mathcal{L}}\}$, where each $F_i \in F$ is a function dependent on a subset of variables $\mathbf{x}_i \in X$ defining the relationship among the variables in $\mathbf{x}_i$. Thus, the function $F_i(\mathbf{x}_i)$ denotes the value for each possible assignment of the variables in $\mathbf{x}_i$. Notably, the dependencies between the variables and the functions generate a bipartite graph, called a factor graph, which is commonly used as a graphical representation of such DCOPs (see Section 2.2.3). In a factor graph, each constraint function $F_i(\mathbf{x}_i)$ is represented by a square node and is connected to each of its associated variable nodes $\mathbf{x}_i$ (denoted by circles) by an individual edge. Hence, $|\mathbf{x}_i|$ is the arity of $F_i(\mathbf{x}_i)$ in this particular graphical representation of DCOP. The nodes (variables and functions[1]) of a factor graph G are being held by a set of agents $A = \{A_1, A_2, \ldots A_k\}$. This mapping of nodes to agents is represented by $\delta : \eta \rightarrow A$. Here, η stands for the set of nodes within the factor graph. Each variable/function is being held by a single agent. Nevertheless, each agent can hold several variables/functions. The corresponding agent acts (i.e. generates and transmits messages) on behalf of the nodes they hold, and is responsible for assigning values to the variables they hold. Within the model, the objective of a DCOP algorithm is to have each agent assign values to its associated variables from their corresponding domains in order to maximize the aggregated global objective function, which eventually produces the value of each variable, X^* (Equation 3.1).

$$X^* = \arg\max_X \sum_{i=1}^{\mathcal{L}} F_i(\mathbf{x}_i) \tag{3.1}$$

For example, Figure 3.1 illustrates the relationship among variables, functions and agents of a factor graph representation of a sample DCOP. Here, we have a set of four variables $X = \{x_0, x_1, x_2, x_3\}$, a set of two functions/factors $F = \{F_0, F_1\}$, and a set of four agents $A = \{A_1, A_2, A_3, A_4\}$. Moreover, $D = \{D_0, D_1, D_2, D_3\}$ is a set of discrete and finite variable domains, each variable $x_i \in X$ can take its value from the domain D_i. In this example, agent A_1 holds a function node F_0 and a variable node x_0. Similarly, nodes F_1 and x_1 are being held by agent A_3. While agents A_2 and A_4 hold variable nodes x_2 and x_3, respectively. In this particular setting, four agents A_1, A_2, A_3 and A_4 participate in the optimization process in order to maximize a global objective function $F(x_0, x_1, x_2, x_3)$. Here, the global objective function is an aggregation of two local functions $F_0(x_0, x_1, x_2)$ and $F_1(x_1, x_2, x_3)$. In the factor graph, F_0 is associated (i.e.

[1]The term function is also known as factor, and they are used interchangeably throughout this thesis.

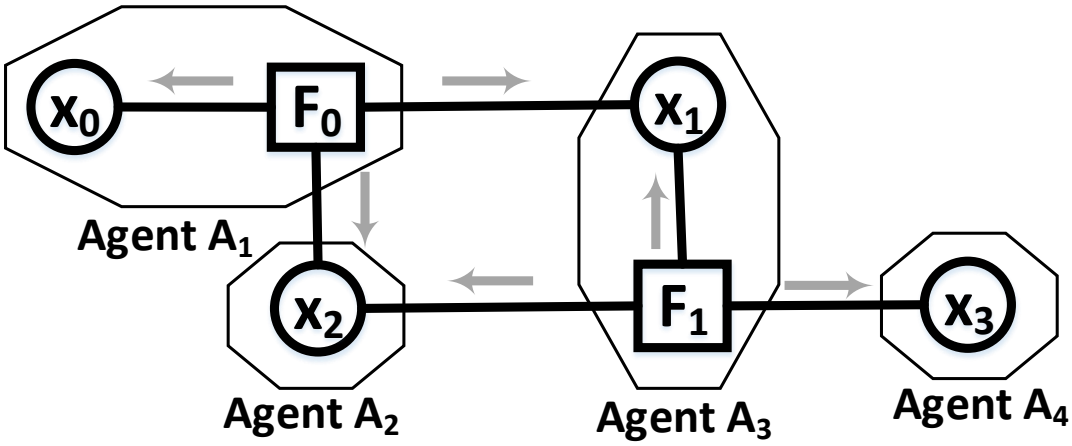


Figure 3.1: In the figure, the same factor graph shown in Figure 2.4 is used to highlight (i.e. grey arrows) the factor-to-variable messages of GDL-based algorithms, each of which requires the maximization operation to be performed.

connected) with three variable nodes, and as such, the arity of the constraint function F_0 is 3. Similar to F_0, the arity of constraint function F_1 is 3 in this particular example.

GDL-based inference algorithms follow a message passing protocol to exchange messages among the nodes of the factor graph (see Section 2.4.2 for more details). Here, both the Max-Sum and BMS algorithms use Equations 3.2 and 3.3 for their message passing. Specifically, the variable and function nodes of a factor graph continuously exchange messages (variable x_i to function F_j (Equation 3.2) and function F_j to variable x_i (Equation 3.3)) to compute an approximation of the impact that each of the agents' actions have on the global objective function by building a local objective function $Z_i(x_i)$. In Equations 3.2−3.4, M_i stands for the set of function nodes connected to variable x_i and N_j represents the set of variable nodes connected to function F_j. Once the function is built (Equation 3.4), each agent picks the value of a variable that maximizes the function by finding $\arg\max_{x_i}(Z_i(x_i))$.

$$Q_{x_i \to F_j}(x_i) = \sum_{F_k \in M_i \setminus F_j} R_{F_k \to x_i}(x_i) \tag{3.2}$$

$$R_{F_j \to x_i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} [F_j(\mathbf{x}_j) + \sum_{x_k \in N_j \setminus x_i} Q_{x_k \to F_j}(x_k)] \tag{3.3}$$

$$Z_i(x_i) = \sum_{F_j \in M_i} R_{F_j \to x_i}(x_i) \tag{3.4}$$

As discussed in Section 2.5.2, due to the potentially large parameter domain size and constraint functions with high arity, the maximization operator of the factor-to-variable message is the main reason GDL-based algorithms can be computationally expensive. This can be visualized from an example where a function node has five variable nodes connected to it, meaning the arity of the function is $n = 5$. Here, we assume each of the variables can take its value from 9 possible options (i.e. states of the domain), implying that the domain size is $d = 9$ for each of the variables. In this case, the function node has to perform 9^5 or $59,049$ operations to generate a message for one of its neighbouring variable nodes. Now, each of the function nodes in a factor graph has to generate and send a single message to each of its neighbours to complete a single round of message passing (Aji & McEliece, 2000; Kschischang et al., 2001). For example, function node F_1 of Figure 3.1 has to send a distinct message (grey arrow) to each of its neighbouring variable nodes x_1, x_2 and x_3. Each of these messages includes the expensive maximization operator. Under such circumstances, it is possible to significantly reduce the computational cost of this step. Meanwhile, it is essential to ensure that this reduction process does not limit the algorithms' applicability, as well as not affecting the solution quality. We deal with the issue that arises from the trade-off in the section that follows.

3.2 The Generic Domain Pruning Technique

$\mathcal{GDP}$ (Algorithm 2) works as a part of Equation 3.3, which represents a function-to-variable message of a GDL-based algorithm, in order to reduce the search space over which the maximization needs to be computed. This algorithm requires as inputs a sending function node $F_j(\mathbf{x}_j)$ whose utility depends on a set of variable nodes $(\mathbf{x}_j)$ associated with it (i.e. neighbours), a receiving variable node $x_i \in \mathbf{x}_j$ and all the incoming messages from the neighbour(s) of F_j apart from the receiving node x_i, denoted as $\mathcal{M}_{\mathbf{x}_j \setminus x_i}$. Finally, $\mathcal{GDP}$ returns a pruned range of values for each state of the domains of the variables over which the maximization operation needs to be performed to generate the message from the function node F_j to the variable node x_i (i.e. $R_{F_j \to x_i}(x_i)$).

In more detail, $\mathbb{S}$ stands for a set $\{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ representing each state of the domains corresponding to $\mathbf{x}_j$ (line 1 of Algorithm 2). This implies that $\mathbb{S}$ is the union $(\cup)$ of those sets of states, each of which corresponds to the domain of a variable in $\mathbf{x}_j$. Line 2 sorts the local utility of the sending function node F_j independently by each state $\mathbf{s_i} \in \mathbb{S}$. This sorting can be carried out at runtime of a message passing algorithm without incurring an additional delay (discussed shortly in *Conjecture* 1). Then the total number of incoming messages received by F_j is represented by $\mathbf{n}$ (line 3). Note that, a complete worked example of $\mathcal{GDP}$ is depicted in Figure 3.2 where we use a part of the factor graph of Figure 3.1 to show a factor-to-variable (i.e. F_1 to x_3) message computation (Figure 3.2(a)), as well as the operation of $\mathcal{GDP}$ on it (Figure 3.2(b)).

Algorithm 2: Generic Domain Pruning- $\mathcal{GDP}(F_j(\mathbf{x}_j), x_i, \mathcal{M}_{\mathbf{x}_j \setminus x_i})$

Input: $F_j(\mathbf{x}_j)$ - Local utility of the sending function node F_j, where $\mathbf{x}_j$ is the set of variable nodes associated with F_j;

$x_i \in \mathbf{x}_j$ - the variable node which is going to receive a message from F_j;

$\mathcal{M}_{\mathbf{x}_j \setminus x_i}$ - Received messages by F_j from all of its neighbouring variable nodes $\mathbf{x}_j$, other than x_i.

Output: Pruned range of values of the states over which maximization needs to be performed to generate the message from F_j to x_i.

Let $\mathbb{S} = \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ be the states corresponding to the domains of $\mathbf{x}_j$

Sort the local utility $F_j(\mathbf{x}_j)$ independently by each state $\mathbf{s_i} \in \mathbb{S}$

$\mathbf{n} \leftarrow |\mathcal{M}_{\mathbf{x}_j \setminus x_i}|$

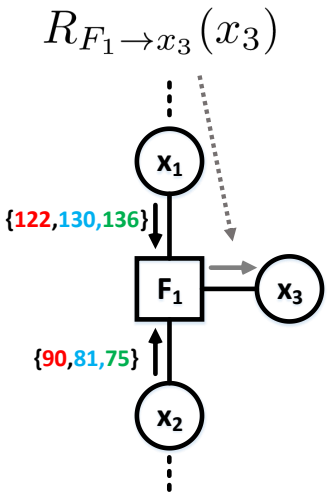
$m \leftarrow \sum_{k=1}^{\mathbf{n}} \max(\mathcal{M}_k)$, where $\mathcal{M}_k \in \mathcal{M}_{\mathbf{x}_j \setminus x_i}$ is one of the $\mathbf{n}$ messages received by F_j

foreach $\mathbf{s_i} \in \mathbb{S}$ **do** `// for each state corresponding to the variables` $\mathbf{x}_j$ `that associate with` F_j

- $\mathcal{V}_i \leftarrow sortedVal_{\mathbf{s}_i}(F_j(\mathbf{x}_j))$
- $p \leftarrow \max(\mathcal{V}_i)$
- $b \leftarrow \sum_{k=1}^{\mathbf{n}} val_p(\mathcal{M}_k)$
- $t \leftarrow m - b$
- $\mathbf{q} \leftarrow binarySearch(\mathcal{V}_i, \lambda)$ *where* $\lambda = \max_c\{c \in \mathcal{V}_i : c \leq (p - t)\}$
- **if** $\mathbf{q} == p - t$ **then**
  - **result** $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}])$
- **else**
  - **result** $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}))$

Utility (cost) table for the function node, $F_1(x_1, x_2, x_3)$

x_1	x_2	x_3	F_1
R	R	R	5
R	R	B	10
R	R	G	12
R	B	R	13
R	B	B	12
R	B	G	35
R	G	R	9
R	G	B	38
R	G	G	11
B	R	R	12
B	R	B	14
B	R	G	38
B	B	R	10
B	B	B	3
B	B	G	9
B	G	R	40
B	G	B	14
B	G	G	13
G	R	R	10
G	R	B	37
G	R	G	12
G	B	R	39
G	B	B	13
G	B	G	14
G	G	R	11
G	G	B	12
G	G	G	4



$Q_{x_1 \to F_1}(x_1)$	$Q_{x_2 \to F_1}(x_2)$	F_1		$R_{F_1 \to x_3}(x_3)$
122	90	5	217	
122	90	10	222	
122	90	12	224	
122	81	13	216	
122	81	12	215	
122	81	35	238	
122	75	9	206	
122	75	38	235	
122	75	11	208	
130	90	12	232	
130	90	14	234	
130	90	38	258	{256,263,258}
130	81	10	221	
130	81	3	214	
130	81	9	220	
130	75	40	245	
130	75	14	219	
130	75	13	218	
136	90	10	236	
136	90	37	263	
136	90	12	238	
136	81	39	256	
136	81	13	230	
136	81	14	231	
136	75	11	222	
136	75	12	223	
136	75	4	215	

(a) Computation of a function-to-variable message (i.e. F_1 to x_3).

Domain pruning of state R during the computation of $R_{F_1 \to x_3}(x_3)$

$M_1 = Q_{x_1 \to F_1}(x_1) = \{122, 130, 136\}$

$M_2 = Q_{x_2 \to F_1}(x_2) = \{90, 81, 75\}$

$V_R = sortedVal_R(F_1(x_1, x_2, x_3)) = \{40, 39, 13, 12, 11, 10, 10, 9, 5\}$

$m = max\ (M_1) + max\ (M_2) = 136 + 90 = 226$

$p = max\ (V_R) = 40; Therefore, b = 130 + 75 = 205$

Given $m = 226$, $t = m - b = 226 - 205 = 21$

Target – find the row that contains utility value $p - t$, if not then closest smaller value of $p - t$.

Here, $p - t = 40 - 21 = 19$

Thus, the desired value ***q*** is *13*, which is in the third row from the top. Use binary search to find this value

Computation for state R

$Q_{x_1 \to F_1}(x_1)$	$Q_{x_2 \to F_1}(x_2)$	F_1	"*Sum*"
130	75	40	245
136	81	39	256
122	81	13	216
130	90	12	232
136	75	11	222
136	90	10	236
130	81	10	221
122	75	9	206
122	90	5	217

"*Max*"

Now, if $q == p - t$, *then the maximum value for R must be found from the rows within the range* $[p, q]$.

Else If $q < p - t$, *then the maximum value for R must be found from the rows within the range [p, q).*

Here, $q < p - t$ (i.e. $13 < 19$), hence we have to look only for the rows within the range $[40, 39]$ or $[40, 13)$, and discard the rest.

(b) Complete operation of $\mathcal{GDP}$ on $R_{F_1 \to x_3}(x_3)$.

Figure 3.2: Worked example of $\mathcal{GDP}$ in computing a factor-to-variable message, F_1 to x_3 or $R_{F_1 \to x_3}(x_3)$, within the factor graph shown in Figure 3.1. In this example, for simplicity, we show that part of the original factor graph which is necessary for this particular message computation. In the figure, red, blue and green coloured values are used to distinguish the domain states R, B and G respectively for each of the variables involved in the computation, and arrows between the nodes of the factor graph are used to indicate the direction of the corresponding messages.

Here, the local utility of the sending function node F_1 is shown in a table at the left side of Figure 3.2(a), which is based on three domain states $\{R, B, G\}$ (for simplicity red, blue and green colours are used to distinguish the values of the states, respectively) and three neighbouring variable nodes x_1, x_2 and x_3. Moreover, the direction of two incoming messages (i.e. $\mathbf{n} = 2$) received by F_1, $\{122, 130, 136\}$ and $\{90, 81, 75\}$, from the variable nodes x_1 and x_2 respectively, are indicated using the black arrows. Then, the grey arrow indicates the desired function-to-variable message $R_{F_1 \to x_3}(x_3) = \{256, 263, 258\}$, and the complete calculation is depicted in a table at the right side of Figure 3.2(a).

At this point, line 4 computes m which is the summation of the maximum values of each of the messages $\mathcal{M}_k \in \mathcal{M}_{\mathbf{x}_j \setminus x_i}$ received by the sending function F_j, other than the receiving variable node x_i. Here, $\mathcal{M}_k$ is one of the $\mathbf{n}$ messages received by F_j. In the worked example of Figure 3.2(b), since the maximum of the received messages $\{122, 130, 136\}$ (i.e. $\mathcal{M}_1$) and $\{90, 81, 75\}$ (i.e. $\mathcal{M}_2$) by F_1 are 136 and 90 respectively, the value of $m = 136 + 90 = 226$. Now, the for loop in lines $5 - 14$ generates the range of the values for each state $\mathbf{s_i} \in \mathbb{S}$ from where we will always find the maximum value for the function F_j, and discard the rest. To this end, the function $sortedVal_{\mathbf{s}_i}(F_j(\mathbf{x}_j))$ gets the sorted value of $\mathbf{s_i}$ from line 2, and stores them in an array $\mathcal{V}_i$ (line 6). Then, line 7 finds p, which is the maximum of the local utility values for the state $\mathbf{s_i}$ (i.e. $\max(\mathcal{V}_i)$). In the worked example, the sorted values of domain state R are stored in $\mathcal{V}_R$, depicted in the right side of Figure 3.2(b). Hence, the value of $p = \max(\mathcal{V}_R) = 40$. Afterwards, line 8 computes b, which is the summation of the corresponding values of p from the incoming messages of F_j (i.e. $val_p(\mathcal{M}_k)$). In the example, the values corresponding to p (i.e. 40) from two incoming messages are 130 and 75, thus the value of $b = 130 + 75 = 205$. This can be seen in the first row of the rightmost table of Figure 3.2(b). The rows related to the computation for the state R are summarized into this table from the rightmost table of Figure 3.2(a), which depicts the complete computation of the function F_1 to variable x_3 message based on domain states R, B and G. Having obtained the value of m and b from lines 4 and 8 respectively, line 9 gets the base case t, which is a subtraction of b from m (i.e. $t = m - b = 226 - 205 = 21$).

Line 10 searches for a value λ in the sorted list $\mathcal{V}_i$ and stores it in a variable $\mathbf{q}$. In this context, λ stands for a value $c \in \mathcal{V}_i$ that is either equal to the value of $p - t$ or immediately smaller than $p - t$. In other words, λ is the maximum of those values in $\mathcal{V}_i$ that are not greater than $p - t$. To this end, we use the binary search method because the list that needs to be searched is already sorted. Now, when the value of $\mathbf{q}$ is equal to $p - t$, the desired maximum value for the state $\mathbf{s}_i$ must always be found by considering the rows corresponding to the values in the range $[p, \mathbf{q}]$, denoted by $prunedRange_{\mathbf{s_i}}([p, \mathbf{q}])$ (lines $11 - 12$)[2]. Otherwise, the value of $\mathbf{q}$ is less than $p - t$, and the desired maximum value for the state $\mathbf{s}_i$ must always be found by considering the rows corresponding to the values

in the range $[p, \mathbf{q})$, denoted by $prunedRange_{\mathbf{s_i}}([p, \mathbf{q}))$ (lines $13 - 14$)[2]. In the worked example of Figure 3.2(b), the value of $p-t$ is 19, given $p = 40$ and $t = 21$. The target is to obtain the value of $\mathbf{q}$ from the list $\mathcal{V}_R$. In the third column of the rightmost table that illustrates the computation for the state R, we see that the value of $\mathbf{q}$ is 13 because this is the closest smaller (or equal) value of 19 (i.e. $p-t$). Since $\mathbf{q}$ is not equal to $p-t$ in this instance, according to lines $13 - 14$ the desired maximization for R must be found by considering the rows corresponding to the values in the range $[40, 13)$ or $[40, 39]$. That means, only considering the top two rows are sufficient to obtain the desired value of R; hence, it is not necessary to consider the remaining 7 rows for this particular instance. To be exact, the value for the state R after maximization is 256, which is obtained from the row corresponding to the local utility value of 39. In this way, $\mathcal{GDP}$ reduces the computational cost of the expensive maximization operator. The grey colour is used to mark the discarded rows of the table. We can see that even for such a small instance, having domain size $d = 3$ and arity $n = 3$, $\mathcal{GDP}$ prunes more than 75% of the search space during the maximization of a state in computing the function-to-variable message.

As argued above, it is important to ensure that combining $\mathcal{GDP}$ with Equation 3.3 does not make the computation of a function-to-variable message prohibitively expensive. Given the sorting operation of line 2 does not incur an additional delay (see *Conjecture* 1), the time complexity of $\mathcal{GDP}$ involves two parts. This includes the for-loop of line 5 and the binary search of line 10. Hence, we determine that the overall time complexity of $\mathcal{GDP}$ is $\mathcal{O}(r \log |\mathcal{V}_i|)$. In this context, r stands for the number of states of the variables' domain associated with the sending function node (line 5). Then, $|\mathcal{V}_i|$ is the size of the array $\mathcal{V}_i$, hence $\log |\mathcal{V}_i|$ is the time complexity to do the binary search of line 10 on $\mathcal{V}_i$. Taken together, $\mathcal{GDP}$ is able to reduce the search space significantly at the expense of a quasi-linear computation cost of its own.

3.3 Theoretical Analysis

In order to ensure the efficacy of $\mathcal{GDP}$, it is necessary to demonstrate that the sorting operation of line 2 of Algorithm 2 can be conducted without incurring any further delay. Moreover, to ensure the accuracy of our approach, it is paramount to prove that the result of each maximization operation is always found within the proposed range of $\mathcal{GDP}$. In the remainder of this section, we provide theoretical analysis of these two vital claims in the form of *Conjecture* 1 and *Lemma* 1, respectively.

Conjecture 1. The sorting operation performed in line 2 of Algorithm 2 does not incur an additional delay during the computation of a factor-to-variable message.

Discussion. The message passing protocol followed by GDL-based DCOP algorithms operates directly on a factor graph (acyclic or cyclic) representation of a DCOP, and it

[2]See *Lemma* 1 and its proof.

can be classified into the following two categories (Aji & McEliece, 2000; Kschischang et al., 2001):

1. *Synchronous message passing approach. A message is sent from a node v on an edge e to its neighbouring node w if and only if all the messages are received at v on edges other than e, summarized for the node associated with e.* This implies that a node in a factor graph is not permitted to send a message to its neighbour until it receives messages from all its other neighbours. Here, for w to be able to generate and send messages to all its other neighbours, it depends on the message from v. To be exact, w cannot compute and transmit messages to its neighbours other than v until it has received all essential messages, including the message from v. In this process, a single round of the message passing process will complete once each of the nodes is able to send a message to all of its neighbours.

2. *Asynchronous message passing approach.* Nodes of a factor graph are initialized randomly, and outgoing messages can be updated at any time and in any sequence. The message passing needs to continue for a number of rounds[3] to either converge or produce an acceptable approximate solution.

Based on both of these versions of the message passing protocol, the three following cases are seen. We are going to illustrate that, for all of these cases, *Conjecture* 1 is true.

- *Case 1:* It is always preferable for acyclic factor graphs to use the synchronous version of message passing (Farinelli et al., 2008; Aji & McEliece, 2000). This is because it requires only one round of message passing to generate the optimal solution in such factor graphs. Therefore, it is redundant to use the asynchronous alternative. In this case, only a very small number of nodes act (i.e. generate and transmit messages) initially, while the rest of the nodes have to wait for their required messages to arrive before they can start generating message(s). Given a sorting operation is not computationally expensive, we propose to apply $\mathcal{GDP}$ to those nodes which are not initially active. Thus, they can utilize the waiting time to complete the sorting operation without incurring an additional delay.

- *Case 2:* A key GDL-based DCOP algorithm, namely Bounded Max-Sum, deals with cyclic factor graph representations of DCOPs by initially removing the cycles from the original factor graph using a pre-processing step. During this step, the agents experience an additional waiting time. Then, it applies the synchronous version of message passing on the transformed acyclic factor graph to provide a bounded approximate solution of the problem (see Section 2.4.2.2). In this case,

[3] A detailed description regarding how many rounds are required is beyond the scope of this work. See Farinelli et al. (2008) for more details.

the sorting operation can be carried out during the agents' waiting time of the preprocessing step. Hence it does not incur an additional delay.

- *Case 3:* The so-called *loopy message passing* (Farinelli et al., 2008) is another way to deal with the cyclic factor graph representation of DCOPs. It uses the asynchronous approach as the message passing protocol. As mentioned above, this version of message passing requires several rounds to either converge or produce an approximate solution. We propose to enforce the fact that the first round must follow the synchronous approach, so that the sorting operation can be completed using the same way as *Case 1.* The following rounds can then proceed with the asynchronous message passing approach without loss of its own characteristics.

□

LEMMA 1. During a function-to-variable message computation, the desired maximum value for a state $\mathbf{s_i} \in \mathbb{S}$ must always be found from the rows corresponding to the values ranging from $\mathbf{q}$ to p.

PROOF. We prove this by contradiction. Assume there exists a row r_a that resides outside the range from which the maximum value for $\mathbf{s_i}$ can be found. As we know, a function-to-variable message depends on two inputs− the local utility table of the function and the incoming messages from its neighbours. In this context, p is the maximum utility corresponding to $\mathbf{s_i}$, and is within our proposed range. Therefore, to be able to find r_a, we have to rely on the only remaining input, that is the incoming messages from the neighbours of the sending function node. To this end, let's consider two parameters from this remaining input. The first is the summation of the maximum values from each of the incoming messages, denoted as m. The second is the summation of values from the incoming messages corresponding to p, denoted as b. Given p is the maximum of the first input, the value $t = m - b$ is significant because this is the maximum difference the remaining input can make. In $\mathcal{GDP}$, the value of $\mathbf{q}$ is chosen in such a way that it covers the difference. As a consequence, there exists no such row as r_a.

□

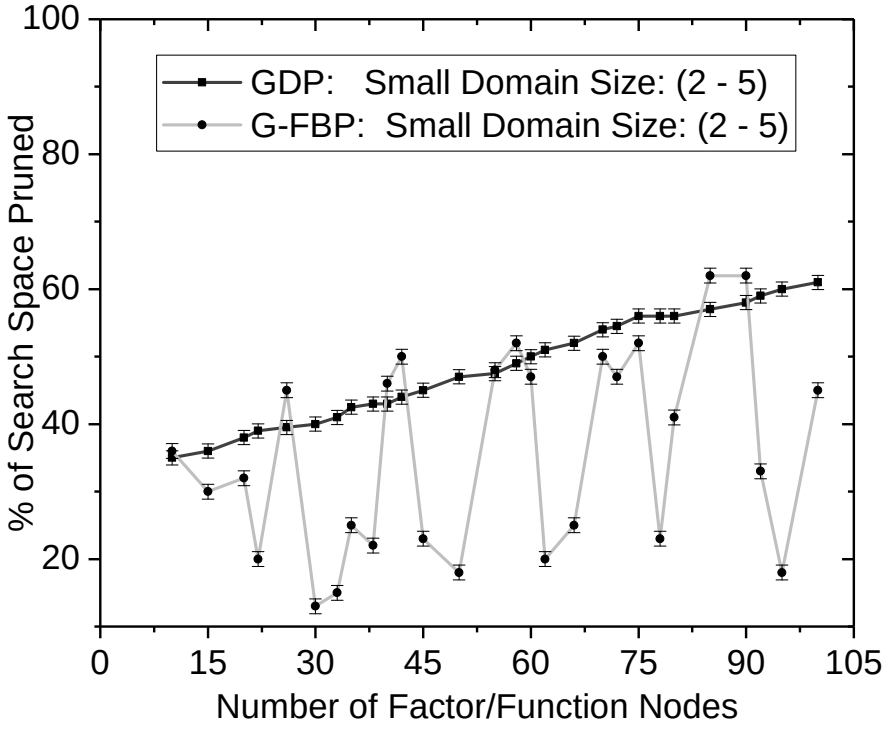
3.4 Empirical Evaluation

We now empirically evaluate how much speed-up can be achieved using $\mathcal{GDP}$ and compare this with the performance of G-FBP[4]. In so doing, we run our experiments on two different types of factor graph representation (i.e. sparse and dense) of a benchmarking graph colouring problem. Specifically, we consider factor graphs having a number of function nodes ranging from 10 to 100, and that each of the factor graphs is generated by randomly connecting a number of variable nodes per function node. Specifically, this

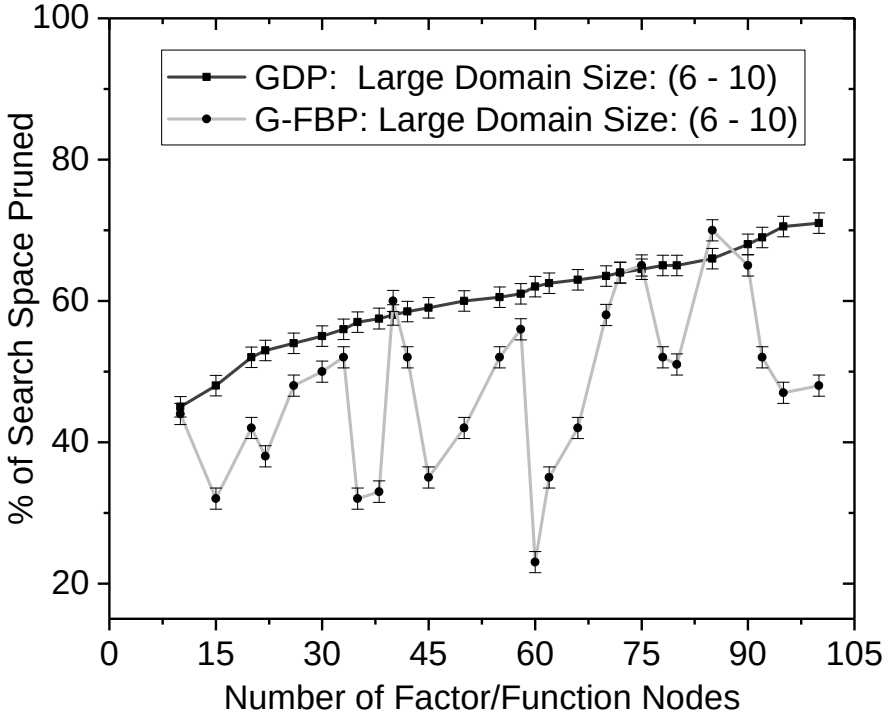
[4]See Sections 2.5.2 and 2.6 for the detailed reasoning behind the selection of this benchmark.

number of variable nodes connected to each function node, termed the arity n of a function, has been chosen based on the following parameters: the value of n for each function node is randomly chosen from the ranges $1-4$ and $5-10$ to generate sparse and dense factor graphs, respectively. Thus, the differences in the arity of the function nodes for two different types of factor graphs are distinguished by the terms sparse and dense in our experiments. Moreover, we categorize domain size d of the variable nodes into two distinct classes. On the one hand, for a setting with *"small domain size"* we consider the size between 2 to 5 for each of the variable nodes in a factor graph. On the other hand, we consider them between 6 to 10 for a setting with *"large domain size"*. This classification has been done in order to observe the performance of $\mathcal{GDP}$ and G-FBP from a very small (e.g. $d^n = 2^3$) to a large (e.g. $d^n = 10^5$) search space. It is worth noting that we make use of the Frodo framework (Léauté et al., 2009) to generate local utility tables (i.e. cost function) for the function nodes of a factor graph. To get the results based on the aforementioned setting, we initially compute the percentage of the search space pruned (i.e. speed-up) by the algorithms for a function node by taking the average of the speed-ups of all the messages sent by that function node. Afterwards, we take the average of the speed-ups of all the nodes in a factor graph. Finally, we report the results of each factor graph averaged over 50 test runs in Figure 3.3, recording standard errors to ensure statistical significance. All of the experiments were performed on a simulator implemented in an Intel *i*7 Quadcore 3.4GHz machine with 16GB of RAM. Note, both the algorithms, $\mathcal{GDP}$ and G-FBP, operate only on the function-to-variable messages of a factor graph in order to reduce the computation cost of the maximization operator. Therefore, experimenting with other typical DCOP parameters and metrics, such as communication cost, message size and completion time, is beyond the scope of the work presented in this chapter (Stranders et al., 2009; Kim & Lesser, 2013).

Figures 3.3(a)−3.3(b) and Figures 3.4(a)−3.4(b) illustrate the performance of $\mathcal{GDP}$ and G-FBP for sparse and dense factor graphs of $10-100$ function nodes, respectively. In the figures, the black lines depict the results of $\mathcal{GDP}$, while the results of G-FBP are shown using the grey lines. More precisely, the black line of Figure 3.3(a) shows the results of $\mathcal{GDP}$ obtained from the factor graphs having variable nodes with *small domain size*. For the same algorithm, the results of the factor graphs having *large domain size* variable nodes are shown using the black line of Figure 3.3(b). It can be clearly seen from those two black lines of both the figures that $\mathcal{GDP}$ always performs better when the variables take their values from a larger domain size, given that the value of the arity n remains identical. Moreover, the performance of $\mathcal{GDP}$ increases steadily with the number of function nodes for both the cases. This trend indicates that $\mathcal{GDP}$ performs correspondingly better when the scale of the factor graph becomes larger. Note that neither all the nodes, nor all the function-to-variable messages experience similar performance from the proposed approach, due to their differences in the content of the utility tables and incoming messages.

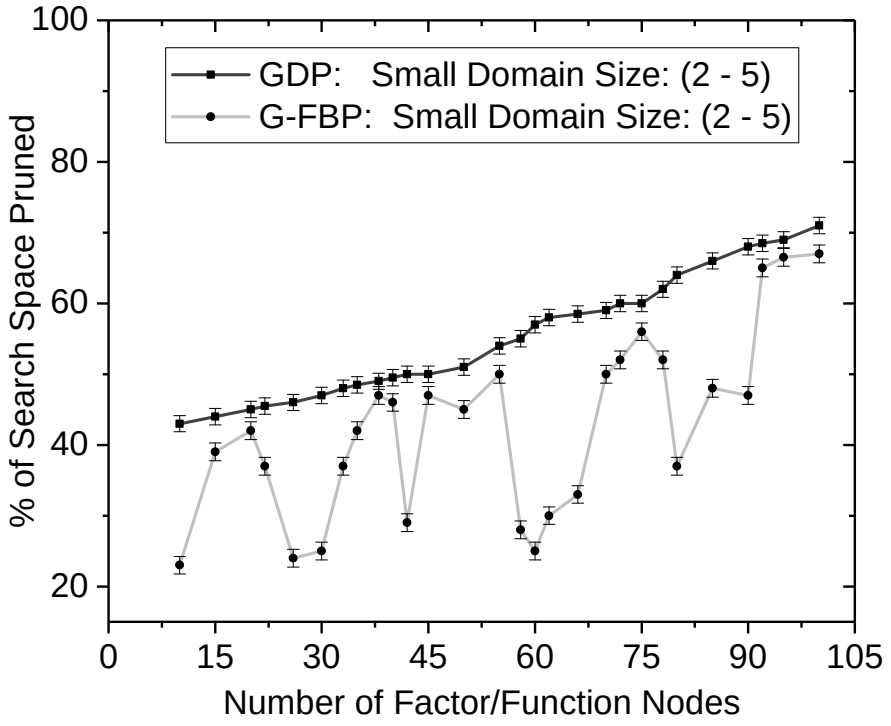


(a) Comparative results for sparse factor graphs having small domain size variable nodes.

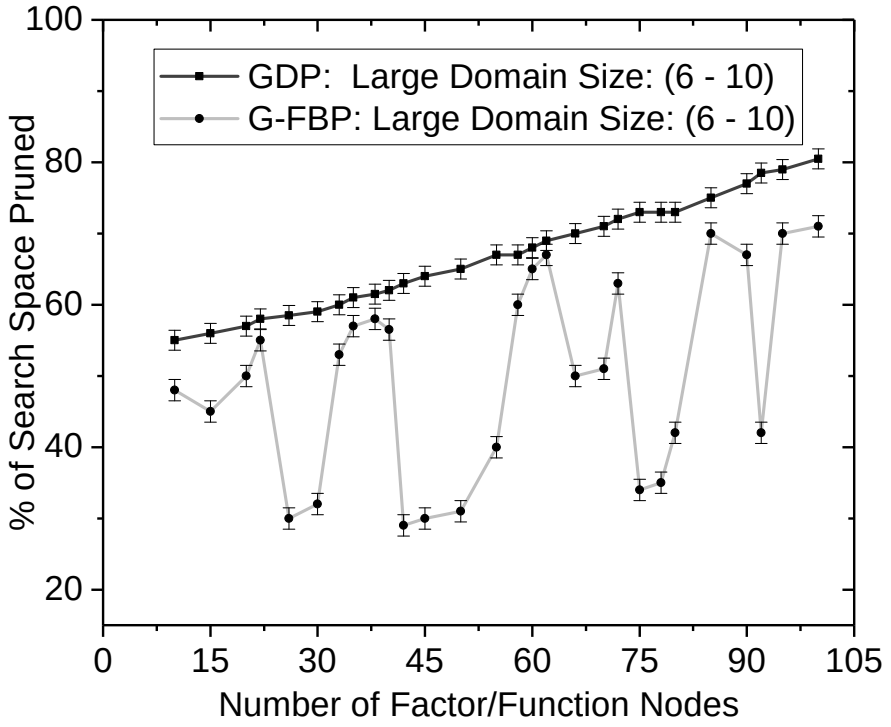


(b) Comparative results for sparse factor graphs having large domain size variable nodes

Figure 3.3: Empirical results: $\mathcal{GDP}$ vs G-FBP− for the factor graph (sparse) representations of different instances of the graph colouring problem. Error bars are calculated using standard error of the mean.



(a) Comparative results for dense factor graphs having small domain size variable nodes.



(b) Comparative results for dense factor graphs having large domain size variable nodes

Figure 3.4: Empirical results: $\mathcal{GDP}$ vs G-FBP$-$ for the factor graph (dense) representations of different instances of the graph colouring problem. Error bars are calculated using standard error of the mean.

In more detail, $\mathcal{GDP}$ running over sparse factor graphs having $10-40$ function nodes and variables with *small domain size* prunes around $33-42\%$ of the search space during the computation of the maximization operation. On the other hand, $\mathcal{GDP}$ prunes around $40-48\%$ in the dense factor graph (Figure 3.4(a)) with a similar setting. This indicates that our approach performs significantly better in dense factor graphs, where the value of arity n is larger, compared to the sparse factor graphs. A similar trend is observed in the larger factor graphs of Figures 3.3(a) and 3.4(a). For instance, having $75-100$ function nodes and *small domain size* variable nodes, $\mathcal{GDP}$ prunes around $55-61\%$ and $60-70\%$ of the search space for sparse and dense factor graphs, respectively. On the other hand, it is observed from the results reported in Figure 3.3 that $\mathcal{GDP}$ always performs better when the domain size of the variable nodes are larger, given the remaining parameters are identical. In the sparse setting, we observe around $60-72\%$ reduction of search space by our approach when applied on the factor graph of $65-100$ function nodes and *large domain size* of the variable nodes (Figure 3.3(b)). Notably, the performance gain from $\mathcal{GDP}$ reaches its maximum level (i.e. $70-81\%$) in the dense factor graph with similar setting (Figure 3.4(b)). This is important because it gives us a clear indication that $\mathcal{GDP}$ is able to prune the maximum amount of search space when the values of n and d becomes larger.

As mentioned already, the grey lines of Figures 3.3(a) $-$ 3.4(b) illustrate the results of G-FBP for the same settings as $\mathcal{GDP}$. It can be seen from the results that the performance obtained from G-FBP fluctuates throughout all the cases. The insight behind this trend is due to the fact that G-FBP is based on an intuition that the maximum value can be found from the partially sorted top $cd^{\frac{n-1}{2}}$ values (see Section 2.5.2 for details). When this presumption is false, it incurs a significant penalty in terms of the computation cost (i.e. search space). As a consequence, although we observe a good reduction of the search space by G-FBP for a number of nodes in a factor graph, its overall performance for a complete factor graph is neither guaranteed, nor consistent. Taken together, the aforementioned results clearly show a significant reduction of search space by $\mathcal{GDP}$ while computing the maximization of function-to-variable messages within a factor graph. In contrast, although G-FBP prunes more of the search space for a number of instances, its overall performance is worse than $\mathcal{GDP}$ most of the time because of the consistency issue. This highlights a key shortcoming of G-FBP is that it is not consistent in pruning the search space, while our approach performs better consistently with the growth of the arity and the domain size. In addition, we run paired t-test on the results of $\mathcal{GDP}$ and G-FBP for all of our experiments, where the obtained p-values are less than 0.05 for each of the cases. Therefore, it is obvious that the results are significant at $p \leq 0.05$.

In the final experiment, we analyse whether $\mathcal{GDP}$ and G-FBP are prohibitively expensive in terms of their execution time. We have to check this because both the algorithms trade this time in order to generate the pruned search space. To this end, Figure 3.5 illustrates this result for both the sparse and dense settings defined in the previous

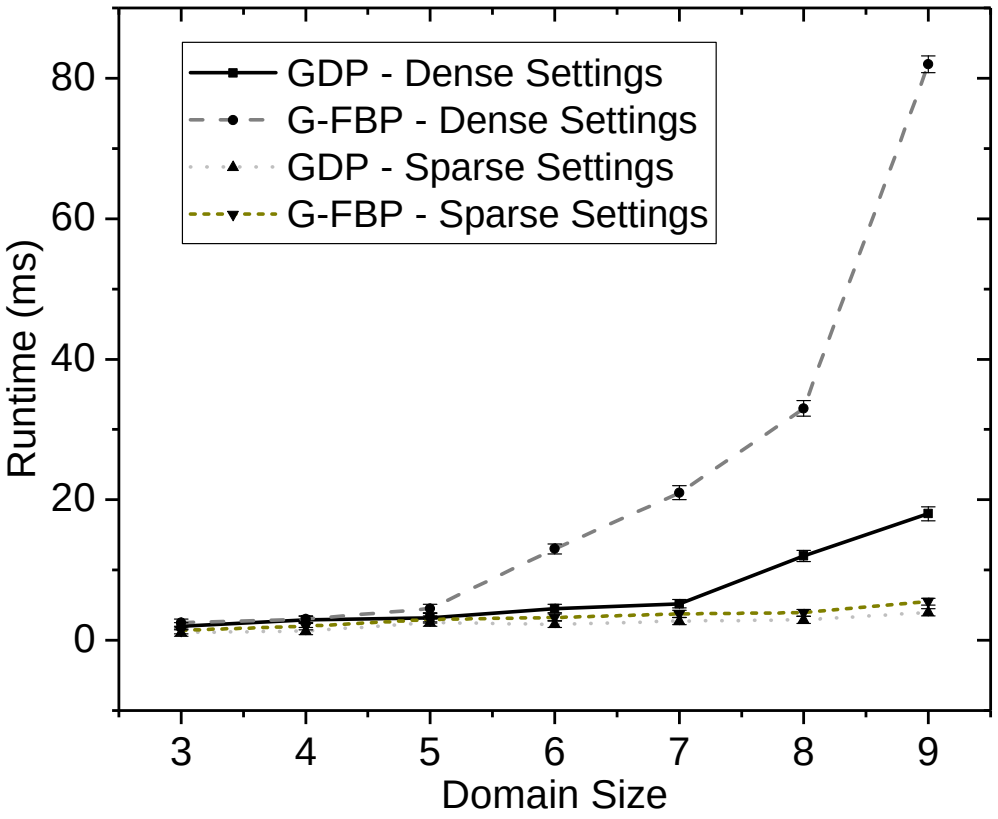


Figure 3.5: Comparative cost of $\mathcal{GDP}$ and G-FBP in terms of their runtime on top of the maximization operator. Error bars are calculated using standard error of the mean.

experiment. The results reported in the figure are generated by taking the average of ten different messages for each of the domain sizes (d) ranging from 3 to 9. On the one hand, it can be clearly seen that the runtime of $\mathcal{GDP}$ (dotted-light-grey line) and G-FBP (short-dash-dark-yellow line) are very small and comparable for all the values of d for sparse settings. On the other hand, when the value of d is more than 5, $\mathcal{GDP}$ (black line) requires comparatively less time than G-FBP (dash-grey line) in the dense settings. Although $\mathcal{GDP}$'s runtime is smaller, none of the algorithms incur such delays that would make them prohibitively expensive to deploy. This is expected because from their complexity analysis we find that both of them require quasi-linear time (see Sections 3.2 and 2.5.2).

3.5 Summary

We presented a new algorithm, $\mathcal{GDP}$ that significantly reduces the computation cost of the maximization operator in the widely used GDL-based DCOP algorithms. We observe a significant reduction in the search space of around $33\% - 81\%$ from our empirical evaluation. This is significant because by reducing the computation cost of the expensive

maximization operator, we are able to accelerate the overall optimization process of this class of DCOP algorithms. Thus, we address the first part of the research challenge **C1**. Moreover, our empirical evidence clearly demonstrates that the performance of $\mathcal{GDP}$ improves with an increase in the parameters upon which the maximization operator acts. Given this, by using $\mathcal{GDP}$, we can now use GDL-based algorithms to efficiently solve large real world DCOPs. In addition, we provide a theoretical proof regarding the accuracy of our approach, which is also applicable on generic DCOP settings as opposed to some previous approaches that tend to be restricted to specific application domains (i.e. the second part of **C1**). Significantly, rather than being a preprocessing step, we have incorporated $\mathcal{GDP}$ into a function-to-variable message of GDL-based algorithms so that they can work jointly. This particular phenomenon provides an opportunity to use existing application dependent approaches on top of $\mathcal{GDP}$ to further reduce the computational cost of the maximization operator.

Chapter 4

Speeding Up the Message Passing Process

In this chapter, we look into the challenge of speeding up the message passing process of GDL-based DCOP algorithms in multi-agent systems (**C2**). To this end, we change the way GDL-based algorithms propagate their messages during the optimization process. That being said, it is challenging to maintain the quality of solution when there is an alteration in the message passing protocol. Considering this trade-off, we propose a generic message passing protocol, Parallel Message Passing ($\mathcal{PMP}$), for GDL-based algorithms that utilizes the benefits of partial centralization, combines clustering with domain pruning, as well as the use of a regression method to determine the appropriate number of clusters for a given scenario. $\mathcal{PMP}$ can actually replace SMP (or its asynchronous alternative) in order to minimize the completion time of such algorithms while maintaining the same solution quality. Therefore, it is possible to increase the scalability of these algorithms in that either they complete the internal operation of a given size of DCOP in a shorter span of time, or they can handle a larger DCOP in the same completion time as a smaller one that uses SMP.

In this chapter, we use SMP as a benchmark in evaluating $\mathcal{PMP}$ because SMP is faster (or in the worst case, equal) to its asynchronous counterpart (see Section 2.5.3). It is worth mentioning that the GDL-based algorithms, which deal with cyclic graphical representations of DCOPs (e.g. BMS[1]and BFMS[1]), initially remove the cycles (i.e. loops) from the original factor graph, then apply the SMP protocol on the acyclic graph to provide a bounded approximate solution of the problem. Comparatively, our protocol can be applied on cyclic DCOPs in the same way. Thus, once the cycles have been removed, $\mathcal{PMP}$ can be applied in place of SMP on the transformed acyclic graph.

[1]BMS has been proposed as a generic approach that can be applied to all DCOP settings, whereas BFMS can only be applied to a specific formulation of task allocation domain.

The remainder of this chapter is structured as follows: Section 4.1 provides detail discussion of how the SMP protocol operates on the corresponding factor graph representation of a DCOP. Then, in Section 4.2, we discuss the technical details of our $\mathcal{PMP}$ protocol with worked examples. Next, we present the performance of our approach through extensive empirical evaluation in Section 4.3. Afterwards, Section 4.4 demonstrates the details and the performance of applying the regression method on $\mathcal{PMP}$. Finally, we conclude this chapter with a summary in Section 4.5.

4.1 Problem Description

To date, the factor graph representation of a DCOP follows the SMP protocol to exchange messages in GDL-based message passing algorithms. Notably, both the Max-Sum and BMS algorithms (two key algorithms based on GDL) use Equations 4.1 and 4.2 for their message passing (see Section 2.4.2 for details of the equations), and they can be directly applied to the factor graph representation of a DCOP. Even though some extensions of Max-Sum and BMS (e.g. FMS, BFMS and BnB FMS) modify these equations slightly (as discussed in Section 2.5.2), the SMP protocol still underpins these algorithms. The reason behind this is that a message passing protocol, by definition, does not depend on how the messages are generated; rather, it ensures when a message should be computed and exchanged (Aji & McEliece, 2000; Kschischang et al., 2001).

$$Q_{x_i \to F_j}(x_i) = \sum_{F_k \in M_i \setminus F_j} R_{F_k \to x_i}(x_i) \tag{4.1}$$

$$R_{F_j \to x_i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} [F_j(\mathbf{x}_j) + \sum_{x_k \in N_j \setminus x_i} Q_{x_k \to F_j}(x_k)] \tag{4.2}$$

$$Z_i(x_i) = \sum_{F_j \in M_i} R_{F_j \to x_i}(x_i) \tag{4.3}$$

Algorithm 3 gives an overview of how SMP operates on a factor graph in a multi-agent system. Here, a number of variable and function nodes of a factor graph F_G are held by a set of agents A. The corresponding agents act (i.e. generate and transmit messages) on behalf of the nodes they hold. Initially, only the variable and the function nodes that are connected to a minimum number of neighbouring nodes in F_G, denoted by $iNodes$, are permitted to send messages to their neighbours. Line 1 of Algorithm 3 finds the set of agents $A_m \in A$ that hold $iNodes$. Specifically, the function $messageUpdate()$ represents the messages sent by the agents on behalf of the permitted nodes they hold to their permitted neighbours in a particular time step within a factor graph. Notably, the SMP protocol ensures that a node, variable or function, within a

Algorithm 3: Overview of the SMP protocol on a factor graph

Input: A set of available agents A that holds the function and the variable nodes of a factor graph F_G that represents a DCOP.

1 Find such agents $A_m \in A$ that hold $iNodes \in F_G$;

2 $messageUpdate(F_G,\ A_m.iNodes,\ iNodes.allNeighbours,\ NULL)$;

3 **while** *each node of* F_G *yet to send messages to all their neighbours* **do**

4 $\quad messageUpdate(F_G,\ A'_m.pNodes,\ pNodes.pNeighbours,\ generatedMessages)$;

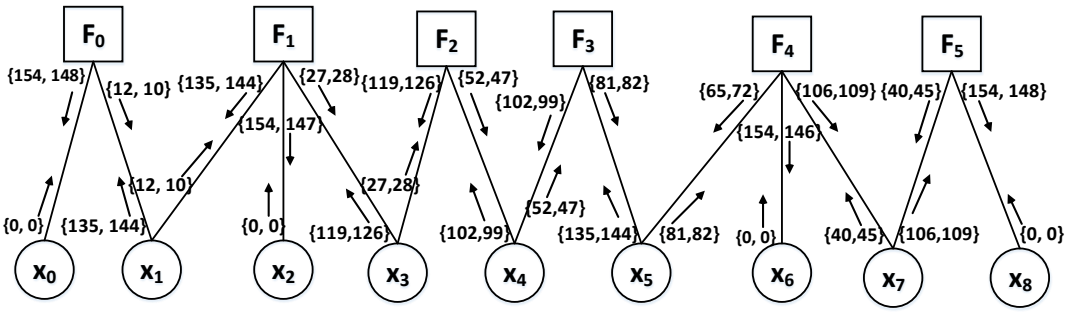
5 $\quad$ **if** *a variable* x_i *receives messages from all of its neighbours within* F_G **then**

6 $\quad\quad x_i$ build a local objective function $Z_i(x_i)$;

7 $\quad\quad$ Agent that holds x_i chooses the value to maximize $Z_i(x_i)$ by finding $\arg\max_{x_i}(Z_i(x_i))$

factor graph cannot generate and transmit a message to its particular neighbour before receiving messages from the rest of its neighbour(s). According to this regulation of SMP, the permitted nodes ($pNodes$) and their corresponding permitted neighbours ($pNodes.pNeighbours$) for a particular time step are determined. At the very first time step, agents A_m on behalf of $iNodes$, also denoted as $A_m.iNodes$, send NULL values to all their neighbouring nodes ($iNodes.allNeighbours$) within F_G (line 2). Now, following the SMP protocol, a set of agents A'_m on behalf of $pNodes$, namely $A'_m.pNodes$, compute messages ($generatedMessages$) using Equation 4.1 or 4.2 for those neighbours ($pNodes.pNeighbours$) they are allowed to send to (line 4). The *while* loop in lines $3-7$ ensures that this will continue until each of the nodes sends messages to all their neighbouring nodes. Within this loop, once a variable x_i receives messages from all of its neighbours, it can build a local objective function $Z_i(x_i)$, and the corresponding agent chooses the value to maximize it by finding $\arg\max_{x_i}(Z_i(x_i))$(lines $5-7$).

Figure 4.1 demonstrates a worked example of how SMP works on the factor graph representation of a DCOP. Here, Equation 4.4 and Equations 4.5−4.8 illustrate two samples of how the variable-to-function (e.g. x_5 to F_4 or $Q_{x_5\to F_4}(x_5)$) and the function-to-variable (e.g. F_4 to x_6 or $R_{F_4\to x_6}(x_6)$) messages are computed based on Equation 4.1 and Equation 4.2, respectively. All the messages are generated considering the local utilities depicted at the bottom of the Figure 4.1 for domain: $\{R, B\}$. During the computation of the messages, red and blue colours are used to distinguish the values of the domain states R and B, respectively. In the former example, apart from the receiving node F_4, the sending node x_5 has only one other neighbouring node (i.e. F_3). Therefore, x_5 only needs to forward the message it received from F_3 to the node F_4 (Equation 4.4). In the latter example, the computation of the message from F_4 to x_6 includes a maximization operation on the summation of the local utility function F_4 and the messages received by F_4 from its neighbours other than x_6 (i.e. x_5, x_7). Given



F_0	x_0	x_1
3	R	R
10	R	B
12	B	R
4	B	B

F_1	x_1	x_2	x_3
5	R	R	R
8	R	R	B
15	R	B	R
9	R	B	B
7	B	R	R
18	B	R	B
4	B	B	R
6	B	B	B

F_2	x_3	x_4
5	R	R
20	R	B
24	B	R
6	B	B

F_3	x_3	x_4
8	R	R
30	R	B
34	B	R
9	B	B

F_4	x_5	x_6	x_7
6	R	R	R
9	R	R	B
25	R	B	R
7	R	B	B
6	B	R	R
27	B	R	B
3	B	B	R
2	B	B	B

F_5	x_7	x_8
12	R	R
40	R	B
45	B	R
13	B	B

Figure 4.1: Worked example of SMP on a sample factor graph representation of a DCOP. In the factor graph, each of the tables represents the corresponding local utility of a function for domain $\{R, B\}$. The values within a curly bracket represent a message computed based on these local utilities, and each arrow indicates the sending direction of the message.

the messages are sent by the corresponding agents on behalf of the nodes they hold, for simplicity, we skip mentioning agents from our worked examples in this chapter.

$$Q_{x_5 \to F_4}(x_5) = R_{F_3 \to x_5}(x_5) = \{81, 82\} \tag{4.4}$$

$$R_{F_4 \to x_6}(x_6) = \max_{\{x_5, x_6, x_7\} \setminus \{x_6\}} [F_4(x_5, x_6, x_7) + (Q_{x_5 \to F_4}(x_5) + Q_{x_7 \to F_4}(x_7))] \tag{4.5}$$

$$= \max_{\{x_5, x_7\}} [\begin{bmatrix} 6 & 9 & 25 & 7 & 6 & 27 & 3 & 2 \end{bmatrix} + \begin{bmatrix} 81 & 81 & 81 & 81 & 82 & 82 & 82 & 82 \end{bmatrix}$$

$$+ \begin{bmatrix} 40 & 45 & 40 & 45 & 40 & 45 & 40 & 45 \end{bmatrix}] \tag{4.6}$$

$$= \max_{\{x_5, x_7\}} \begin{bmatrix} 127 & 135 & 146 & 133 & 128 & 154 & 125 & 129 \end{bmatrix} \tag{4.7}$$

$$= \{154, 146\} \tag{4.8}$$

$$Z_1(x_1) = R_{F_0 \to x_1}(x_1) + R_{F_1 \to x_1}(x_1) = \{12, 10\} + \{135, 144\} = \{147, 154\} \tag{4.9}$$

In the example of Figure 4.1, initially, only x_0, x_2, x_6 and x_8 (i.e. $iNodes$) can send messages to their neighbours at the very first time step (line 2 of Algorithm 3). According to the SMP protocol, in time step 2, only F_0 and F_5 can generate and send messages to x_1 and x_7, respectively. It is worth mentioning that, despite receiving a message from x_2 and x_6 respectively in time step 1, F_1 and F_4 cannot send messages in time step 2 as they

have to receive messages from at least two of their three neighbours, according to the regulation of SMP (line 4). Hence, F_1 has to wait for the message from x_1 to be able to generate and send a message to x_3. Similarly, F_1 cannot generate a message for x_2 until it receives messages from x_1 and x_3. Subsequently, x_3 cannot send a message to F_1 until it receives the message from F_2. In this process, a variable x_1 can only build its local objective function $Z_1(x_1)$ when it receives messages from all of its neighbours F_0 and F_1 (Equation 4.9), and this is common for all the variables. In Equation 4.9, from the value $\{R, B\} = \{147, 154\}$ generated by $Z_1(x_1)$ for the variable x_1, the holding agent of x_1 chooses $B = 154$, that is $\arg\max_{x_1}(Z_1(x_1))$. Following the SMP protocol, the complete message passing procedure will end after each node receives messages from all of its neighbours. These are the dependencies we discussed in Section 2.5.3, which make GDL-based algorithms less scalable in practice for large real world problems. Formally, the total time required to complete the message passing procedure for a particular factor graph can be termed as the completion time T, and the ultimate objective is to reduce the completion time while maintaining the same solution quality from a GDL-based message passing algorithm. To address this issue, we introduce a new message passing protocol in the following section.

4.2 The Parallel Message Passing Protocol

$\mathcal{PMP}$ uses a means similar to that of its SMP counterpart in computing messages. For example, Max-Sum messages are used when $\mathcal{PMP}$ is applied to the Max-Sum algorithm, and FMS messages are used when it is applied to the FMS algorithm. Even so, $\mathcal{PMP}$ reduces the completion time by splitting the factor graph into a number of clusters (Definition 4.1), and independently running the message passing[2] on those clusters in parallel. As a result of this, the average waiting time of the nodes during the message passing is lessened. In particular, the completion time of $\mathcal{PMP}$ is reduced to $\frac{T_{smp}}{\mathcal{N}_\mathcal{C}}$; where T_{smp} is the completion time of the algorithm that follows the SMP protocol, and $\mathcal{N}_\mathcal{C}$ is the number of clusters. However, $\mathcal{PMP}$ ignores inter cluster links (i.e. messages) during the formation of clusters. Hence, it is not possible to obtain the same solution quality as the original algorithm by executing only one round of message passing. This is why $\mathcal{PMP}$ requires two rounds of message passing and an additional intermediate step. The role of the intermediate step is to generate the ignored messages (Definition 4.2) for the split node(s) of a cluster, so that the second round can use these as initial values for those split node(s) to compute the same messages as the original algorithm. To be precise, a representative agent (or a cluster head) takes the responsibility of performing the operation of the intermediate step for the corresponding cluster. To make it possible, we assume that each of the cluster heads retains full knowledge of that cluster, and it can communicate with its neighbouring clusters, making $\mathcal{PMP}$ a partially centralized

[2]It can either be SMP or its asynchronous alternative, without loss of generality, we use SMP from now on (see Section 2.5.3 for details).

Algorithm 4: Overview of the $\mathcal{PMP}$ protocol on a factor graph

Input: A set of available agents A that holds the function and the variable nodes of a factor graph F_G that represents a DCOP.

1 Find an agent $A_c \in A$ that holds the cluster initiator function node $firstFunction$;

2 $\{c_1, c_2, \ldots, c_{\mathcal{N}_\mathcal{C}}\} \leftarrow distributeNodes(F_G, A_c)$, each cluster c_i is a sub-factor graph of F_G;

3 Find such agents $A_m \in A$ that hold $iNodes_{c_i} \in c_i$;

4 $messageUpdate(c_i,\ A_m.iNodes_{c_i},\ iNodes_{c_i}.allNeighbours,\ NULL)$;

5 **while** *All nodes of cluster c_i, having only one neighbouring cluster, yet to send messages to all their neighbouring nodes* **do**

6 $\quad messageUpdate(c_i,\ A'_m.pNodes,\ pNodes.pNeighbours,\ generatedMessages)$

7 A representative agent from each cluster c_i computes the ignored values $ignVal(c_i)$ for that cluster;

8 $messageUpdate(c_i,\ A_m.iNodes_{c_i},\ iNodes_{c_i}.allNeighbours,\ ignVal(c_i))$;

9 **while** *All nodes of each of the clusters c_i yet to send messages to all their neighbours* **do**

10 $\quad messageUpdate(c_i,\ A'_m.pNodes,\ pNodes.pNeighbours,\ generatedMessages)$;

11 $\quad$ **if** *a variable x_i receives messages from all of its neighbours within c_i* **then**

12 $\quad\quad x_i$ build a local objective function $Z_i(x_i)$;

13 $\quad\quad$ Agent that holds x_i chooses the value to maximize $Z_i(x_i)$ by finding $\arg\max_{x_i}(Z_i(x_i))$

approach. As a consequence of two rounds of message passing and an intermediate step, the total completion time of $\mathcal{PMP}$ (i.e. T_{pmp}) becomes $2 \times \frac{T_{smp}}{\mathcal{N}_\mathcal{C}} + T_{intm}$, where T_{intm} is the time required to complete the intermediate step. As the sizes of the clusters can be different in $\mathcal{PMP}$, a more precise way to compute T_{pmp} is through Equation 4.10. Here, $T_{c_{largest}}$ stands for the time required to complete the message passing process of the largest cluster in $\mathcal{PMP}$. Having discussed how to compute the completion time of $\mathcal{PMP}$, we explain the details of our proposed algorithm in the remainder of the section.

$$T_{pmp} = 2 \times T_{c_{largest}} + T_{intm} \tag{4.10}$$

4.2.1 Algorithm Overview

Algorithm 4 gives an overview of $\mathcal{PMP}$. Similar to SMP, it works on a factor graph F_G, and the variable and the function nodes of F_G are being held by a set of agents A. To form the clusters in a decentralized manner, $\mathcal{PMP}$ finds an agent $A_c \in A$ that holds a special function node ($firstFunction$), which initiates the cluster formation procedure (line 1). Specifically, $firstFunction$ is a function node that shares variable(s)

with only one function node. As $\mathcal{PMP}$ operates on acyclic or transformed acyclic factor graphs, such node(s) will always be found. Now, each agent that holds a function node maintaining this property broadcasts an initiator message, and any agent can be picked if more than one agent is found. Subsequently, in line 2, agent A_c initiates the procedure $distributeNodes(F_G, A_c)$ that distributes the nodes of F_G to the clusters $\{c_0, c_1, \ldots, c_{\mathcal{NC}}\}$ in a decentralized way, and the detail of the cluster formation procedure will be explained shortly in Algorithm 5. Note that, in $\mathcal{PMP}$ all the operations within each cluster are performed in parallel.

After the cluster formation procedure has completed, $\mathcal{PMP}$ starts the first round of message passing (lines $3-6$). Line 3 finds the set of agents $A_m \in A$ that hold the variable and function nodes $iNodes_{c_i}$ that are connected to the minimum number of neighbouring nodes within each cluster c_i. Then, $messageUpdate()$ of line 4 represents those messages with NULL values sent by A_m on behalf of $iNodes_{c_i}$, also denoted as $A_m.iNodes_{c_i}$, to all their neighbouring nodes ($iNodes_{c_i}.allNeighbours$) within the cluster c_i. Afterwards, following the same procedure as SMP, a set of nodes $A'_m.pNodes$ generate the messages ($generatedMessages$) for the neighbours ($pNodes.pNeighbours$) they are allowed to send messages to (line 6). However, unlike SMP where the message passing procedure operates on the entire F_G, $\mathcal{PMP}$ executes the first round of message passing on the clusters having only one neighbouring cluster in parallel (line 5). This is because, in the first round, it is redundant to run message passing on the cluster having more than one neighbouring cluster, as a second round will re-compute the messages (see the explanation in Section 4.2.3). The *while* loop in lines $5-6$ ensures that this procedure will continue until each of the nodes sends messages to all its neighbouring nodes within the participating clusters of the first round. Next, a representative agent from each cluster c_i computes the values (Definition 4.2) ignored during the cluster formation procedure for that particular cluster (line 7). Note that these ignored values, represented by $ignVal(c_i)$, are the same values for those edges, should we run SMP on the complete factor graph F_G.

Finally, the second round of message passing is started on all of the clusters in parallel, by considering $ignVal(c_i)$ as initial values for those ignored edges (line 8). Similar to the first round, the *while* loop ensures that this procedure will continue until each of the nodes sends messages to all its neighbouring nodes within c_i (lines $9-13$). Within the second round of message passing, once a variable x_i receives messages from all of its neighbours within c_i, it can build a local objective function $Z_i(x_i)$. Then, the corresponding agent chooses the value that maximizes it by finding $\arg\max_{x_i}(Z_i(x_i))$ (lines $12-13$). By considering the $ignVal(c_i)$ values in the second round, $\mathcal{PMP}$ generates the same solution quality as SMP. We give a more detailed description of each part of $\mathcal{PMP}$ with a worked example in the remainder of this section. To be exact, Section 4.2.2 concentrates on the cluster formation and the message passing procedure. Then, Section 4.2.3 presents the intermediate step. Finally, Section 4.2.4 ends this section with a complete comparative example, SMP versus $\mathcal{PMP}$, in terms of the completion time.

Definition 4.1. *(Cluster, Neighbouring Clusters and Split Node).* A cluster c_i is a sub factor graph of a factor graph F_G. Two clusters c_i and c_j are neighbours if and only if they share a common variable node (i.e. split node) x_p. For instance, c_1 and c_2 of Figure 4.2 are two sub factor graphs of the entire factor graph shown in Figure 4.1. Here, c_1 and c_2 are neighbouring clusters as they share variable x_3 as a split node.

Definition 4.2. *(Ignored Values of a Cluster, $ignVal(c_i)$).* The value(s) overlooked, through the split node(s) of each cluster c_i, during the first round of message passing. In other words, these are the incoming messages through the split node(s), should the SMP protocol have been followed. The intermediate step of $\mathcal{PMP}$ takes the responsibility of computing these ignored values, so that they can be used in the second round in order to obtain the same solution from an algorithm as its SMP counterpart. In the example of Figure 4.2, the intermediate step recovers $\{R, B\} = \{119, 126\}$ for the split node x_3 of cluster c_1, which is going to be used as an initial value for x_3 in the second round of message passing instead of $\{R, B\} = \{0, 0\}$.

4.2.2 Cluster Formation and Message Passing

$\mathcal{PMP}$ operates on a factor graph F_G of a set of variables X and a set of functions F. Specifically, lines $1 - 12$ of Algorithm 5 generate $\mathcal{N}_\mathcal{C}$ *clusters* by splitting F_G. In the process, lines $1 - 2$ compute the maximum number of function nodes per cluster ($\mathcal{N}$), and associated variable nodes of a corresponding function goes to the same cluster. Now, line 3 gets a special function node $firstFunction$, which is a function node that shares variable node(s) with only one function node (i.e. $min\_nFunction(F)$). Any node can be chosen in case of a tie. Then, line 4 initializes the variable "*node*" with the chosen $firstFunction$, which is the first member node of the cluster c_1. The *for* loop of lines $5 - 12$ iteratively adds the member nodes to each cluster c_i. In order to do this in a decentralized manner, a special variable *count* is used as a token to keep track of the current number of function nodes belonging to c_i. When a new node added to c_i is held by a different agent, the variable *count* is passed to the new holding agent. The *while* loop (lines $7 - 11$) iterates as long as the member *count* for a cluster c_i remains less than the maximum nodes per cluster ($\mathcal{N}$), then the new node becomes a member of the next cluster. In the worked example of Figure 4.2, we use the same factor graph shown in Figure 4.1 that consists of 6 function nodes $\{F_0, F_1, \ldots, F_5\}$ and 9 variable nodes $\{x_0, x_1, \ldots, x_8\}$. Here, F_0 and F_5 satisfy the requirements to become the $firstFunction$ node as both of them share variable nodes with only one function node (F_1 and F_4, respectively). We pick F_0 randomly as the $firstFunction$ which eventually becomes the first node for the first cluster c_1; therefore, the holding agent of node F_0 now holds the variable *count* as long as the newly added nodes are held by the same agent. Assume, the number of clusters ($\mathcal{N}_\mathcal{C}$) for this example is 3: c_1, c_2 and c_3 (see Section 4.4 for more details about the appropriate number of clusters). In that case, each cluster can retain a maximum of 2 nodes (functions). According to the cluster formation process

Algorithm 5: Parallel Message Passing

Data: A factor graph, F_G consists a set of variables, $X = \{x_1, x_2, \ldots, x_m\}$ and a set of functions $F = \{F_1, F_2, \ldots, F_{\mathcal{L}}\}$

$\mathcal{N}_{\mathcal{F}} \leftarrow |F|$

$\mathcal{N} \leftarrow \frac{\mathcal{N}_{\mathcal{F}}}{\mathcal{N}_{\mathcal{C}}}$

$firstFunction \leftarrow min\_nFunction(F)$

$node \leftarrow firstFunction$

for $i \leftarrow 1$ ***to*** $\mathcal{N}_{\mathcal{C}}$ **do** `// Cluster formation`

  $count \leftarrow 0$

  while $count < \mathcal{N}$ **do** `// distribute the nodes to each cluster`

    $c_i.member() \leftarrow node$

    $c_i.member() \leftarrow adj(node)$

    $node \leftarrow adj(adj(node))$

    $count \leftarrow count + 2$

  $c_i.member() \leftarrow node$

foreach *cluster* $c_i \in \mathcal{S}_{\mathcal{N}}$ *in* $\mathbb{PARALLEL}$ **do** `// First round of message passing`

  $\forall Q(c_i) \leftarrow \emptyset$

  $\forall R(c_i) \leftarrow \emptyset$

  $Max - Sum(c_i)$: *Message Passing Only* `// Equation 4.1 and Equation 4.2`

for $i \leftarrow 1$ ***to*** $\mathcal{N}_{\mathcal{C}}$ *in* $\mathbb{PARALLEL}$ **do** `// Intermediate step: call Algorithm 6`

  $ignVal(c_i) \leftarrow intermediateStep(c_i)$

for $i \leftarrow 1$ ***to*** $\mathcal{N}_{\mathcal{C}}$ *in* $\mathbb{PARALLEL}$ **do** `// Second round of message passing`

  $(\forall Q(c_i) \setminus Q_{ignEdge}(c_i)) \leftarrow \emptyset$

  $Q_{ignEdge}(c_i) \leftarrow ignVal(c_i)$

  $\forall R(c_i) \leftarrow \emptyset$

  $Max - Sum(c_i)$: *Complete* `// Equation 4.1, Equation 4.2 and Equation 4.3`

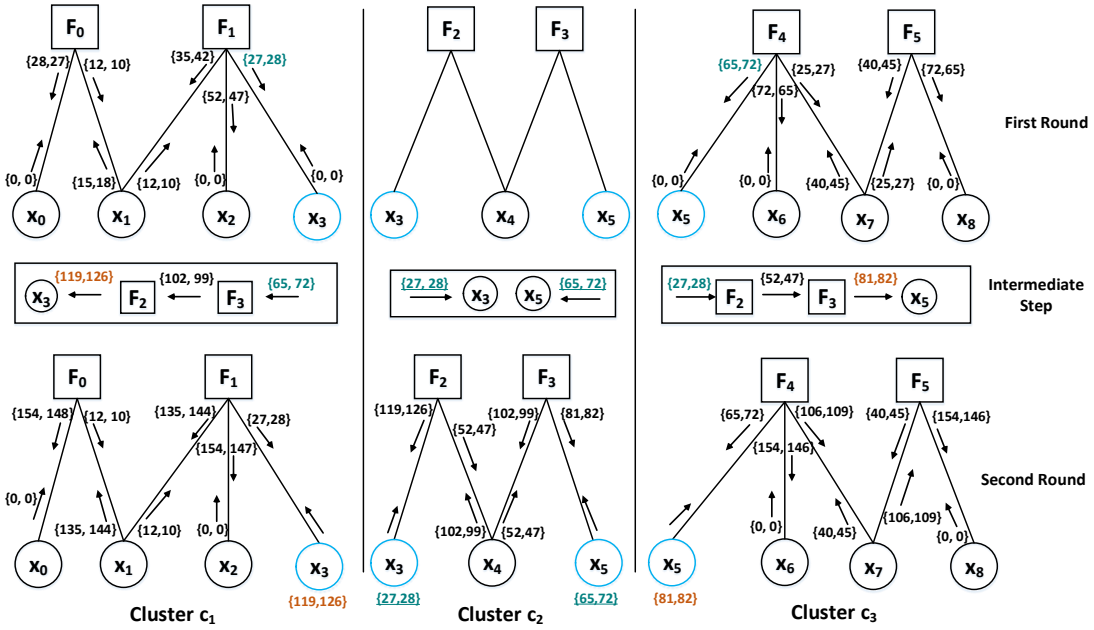


Figure 4.2: Worked example of $\mathcal{PMP}$ (participating clusters: first round - (c_1, c_3) and second round - (c_1, c_2, c_3)) on the same factor graph and local utility as Figure 4.1. In this figure, blue circles represent split variables for each cluster and coloured messages show the ignored values ($ignVal()$) recovered during the intermediate step, where yellow messages require synchronous computations but green underlined ones are ready after the first round.

of $\mathcal{PMP}$, the sets of function nodes $\{F_0, F_1\}$, $\{F_2, F_3\}$ and $\{F_3, F_4\}$ belong to the clusters c_1, c_2 and c_3, respectively. Moreover, c_1 and c_2 are neighbouring clusters as they share a common split node x_3. Similarly, split node x_5 is shared by the neighbouring clusters c_2 and c_3. At this point, the first of the two rounds of message passing initiates as the cluster formation procedure has completed.

The *for* loop in lines $13-16$ acts as the first round of message passing. This involves only computing and sending the variable-to-function (Equation 4.1) and the function-to-variable (Equation 4.2) messages within clusters ($\mathcal{S}_{\mathcal{N}}$) having only a single neighbouring cluster in parallel. It can be seen from our worked example of Figure 4.2 that among the clusters c_1, c_2 and c_3, only c_2 has more than one neighbouring clusters. Therefore, only c_1 and c_3 will participate in the first round of message passing. Unlike the first round, all the clusters participate in the second round of message passing in parallel (lines $19-23$). In the second round, instead of using the null values (i.e. predefined initial values) for initializing all the variable-to-function messages, we exploit the recovered ignored values (Definition 4.2) from the intermediate step (lines $17-18$) as initial values for the split variable nodes, as shown in line 21. Here, all the ignored messages from the split nodes of a cluster c_i are denoted as $Q_{ignEdge}(c_i)$. The rest of the messages are then initialized as null (lines $20, 22$). Here, $\forall Q(c_i)$ and $\forall R(c_i)$ represent all the variable-to-function and the function-to-variable messages within a cluster c_i, respectively. For

example, in cluster c_3 all the messages are initialized as zeros for the first round of message passing. Therefore, during the first round, the variable nodes x_5 and x_8 start the message passing with the values {0, 0} and {0, 0} to the function nodes F_4 and F_5 respectively in Figure 4.2. However, in the second round, split node x_5 starts with the value {81, 82} instead of {0, 0} to the function node F_4. Note that this value {81, 82} is the ignored value for the split node x_5 of cluster c_3 computed during the intermediate step of $\mathcal{PMP}$. Significantly, this is the same value transmitted by the variable node x_5 to the function node F_4 when we follow the SMP protocol (see Figure 4.1), which ensures the same solution quality from both the protocols. We describe this intermediate step of $\mathcal{PMP}$ shortly. Finally, $\mathcal{PMP}$ will converge with Equation 4.3 by computing the value $Z_i(x_i)$ and hence finding $\arg\max_{x_i}(Z_i(x_i))$.

4.2.3 Intermediate Step

A key part of $\mathcal{PMP}$ is the intermediate step (Algorithm 6). It takes a cluster (c_i) provided by line 18 of Algorithm 5 as an input, and returns the ignored values (Definition 4.2) for each of the ignored links of that cluster. A representative of each cluster c_i (cluster head $\mathfrak{ch}_i$) performs the operation of the intermediate step for that cluster. Note that each cluster head operates in parallel. Initially, each cluster head needs to receive the *StatusMessages* from the rest of the cluster heads (line 1 of Algorithm 6). Each *StatusMessage* contains the factor graph structure of the sending cluster along with the utility information. Notably, the *StatusMessages* can be formed and exchanged during the time of the first round, thus it does not incur an additional delay. The *for* loop in lines $2-14$ computes the ignored values for each of the split nodes $\mathcal{S}_j \in \mathcal{S}$ ($where,\ \mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k\}$) of the cluster c_i by generating a Dependent Acyclic Graph, $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ (Definition 4.3). In addition to *StatusMessages*, a cluster head also requires a factor to split variable message (M_r) from each of the participating clusters of the first round. This is significant, as only clusters with one neighbouring cluster can participate in the first round, and the M_r message is prepared for the split node $\mathcal{S}_j$ of that neighbouring cluster. The content of M_r will not change as the participating cluster of the first round has no other clusters on which it depends. As a consequence, if a neighbouring cluster of c_i has participated in the first round, the Dependent Acyclic Graph $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ for $\mathcal{S}_j$ comprises only one edge having the message M_r. In more detail, c_p stands for the neighbouring cluster of c_i that shares the split node $\mathcal{S}_j$ (i.e. $adjCluster(c_i, \mathcal{S}_j)$), and the variable $dCount_{c_p}$ holds the value of the total number of clusters adjacent to c_p obtained from the function $totalAdjCluster(c_p)$ (lines $3-4$). If the cluster c_p has no cluster to depend on apart from c_i (i.e. c_p has participated in the first round of message passing), there is no need for further computation as the ignored value for $\mathcal{S}_j$ (i.e. $\mathcal{S}_j.values$) is immediately ready ($READY.\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$) after the first round (lines $6-7$). Here, the function $append(\mathcal{S}_j.values)$ appends the ignored value for $\mathcal{S}_j$ to $ignVal(c_i)$.

Algorithm 6: intermediateStep(Cluster c_i)

Input: A set of clusters, $\mathcal{C} = \{c_1, c_2, \ldots, c_{\mathcal{N}_\mathcal{C}}\}$ with their corresponding cluster heads, $\mathcal{CH} = \{\mathfrak{ch}_1, \mathfrak{ch}_2, \ldots, \mathfrak{ch}_{\mathcal{N}_\mathcal{C}}\}$

Output: Ignored values $ignVal(c_i)$ for the set of k split nodes $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k\}$ of cluster c_i

$\mathfrak{ch}_\mathfrak{i} \leftarrow StatusMessage(\forall \mathcal{CH} \setminus \mathfrak{ch}_\mathfrak{i}) \cap M_r$ // required utility and messages received by $\mathfrak{ch}_\mathfrak{i}$

for $j \leftarrow 1$ ***to*** k **do**

  $c_p \leftarrow adjCluster(c_i, \mathcal{S}_j)$

  $dCount_{c_p} \leftarrow totalAdjCluster(c_p)$

  if $dCount_{c_p} == 1$ **then** // Cluster having only one neighbouring cluster

    $\mathcal{S}_j.values \leftarrow READY.\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$

    $ignVal(c_i) \leftarrow append(\mathcal{S}_j.values)$

  else if $dCount_{c_p} > 1$ **then** // Cluster having more than one neighbouring cluster

    $dNode \leftarrow adjNode(\mathcal{S}_j, c_p)$

    while $dNode \neq \emptyset$ **do** // Formation of dependent acyclic graph for split node $\mathcal{S}_j$

      $\mathcal{D}_\mathcal{G}(\mathcal{S}_j) \leftarrow dNode$

      $dNode \leftarrow adjNode(dNode)$

    $\mathcal{S}_j.values \leftarrow sync(\mathcal{D}_\mathcal{G}(\mathcal{S}_j))$ // synchronous operation (Equation 4.14) on each edge of $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$

    $ignVal(c_i) \leftarrow append(\mathcal{S}_j.values)$

return $ignVal(\mathfrak{c}_\mathfrak{i})$

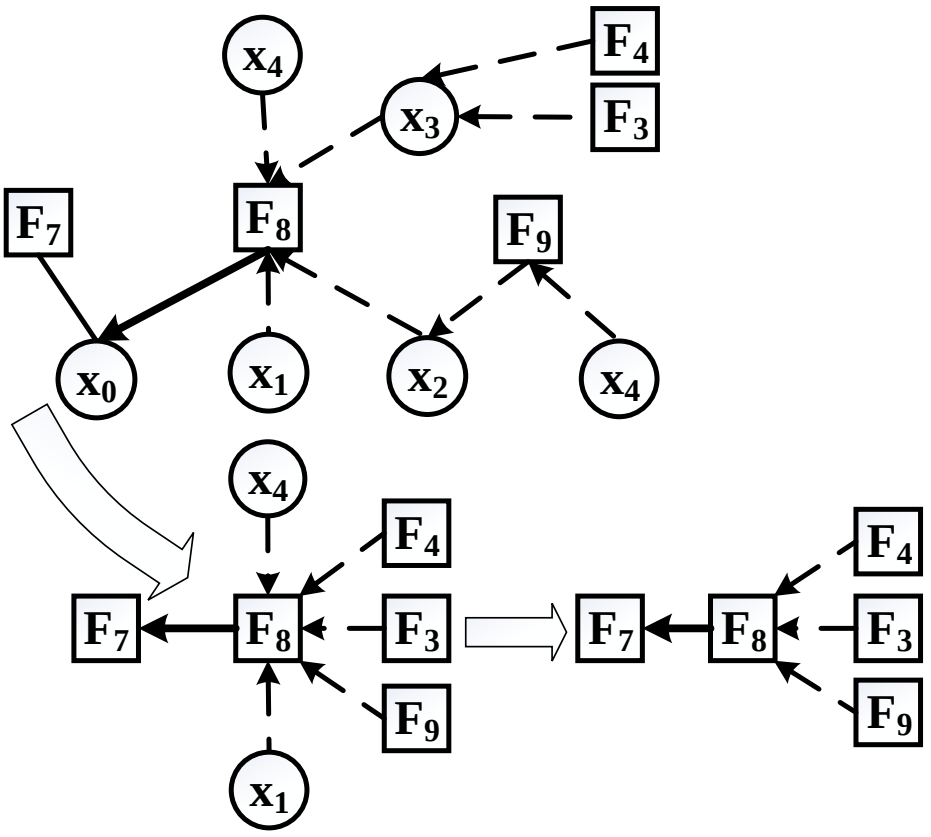


Figure 4.3: Single computation within the intermediate step. In the figure, directed dashed arrows indicate the dependent messages to generate the desired message from F_8 to x_0 or F_8 to F_7 (directed straight arrows).

On the other hand, if the cluster c_p has other clusters to depend on, further computations in the graph are required. This creates the need to find each node of that graph $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ (lines $8-14$). Line 9 initializes the first function node $dNode$ of $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$, which is connected to the split node $\mathcal{S}_j$ and member of the cluster c_p (i.e. $adjNode(\mathcal{S}_j, c_p)$). The *while* loop (lines $10-12$) repeatedly forms that graph through extracting the adjacent nodes from the first selected node, $dNode$. Finally, synchronous executions (explained shortly) from the farthest node to the start node (i.e. split node $\mathcal{S}_j$) of $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ produce the desired value, $\mathcal{S}_j.values$ for $\mathcal{S}_j$ (line 13), which eventually becomes the ignored value for that split node of the cluster c_i (line 14). This value will be used as an initial value during the second round of message passing for the corresponding split node.

Definition 4.3. *(Dependent Acyclic Graph, $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$).* A $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ is an acyclic directed graph for a split node $\mathcal{S}_j$ of a cluster c_i from the furthest node within the factor graph F_G from the node $\mathcal{S}_j$ towards it. Note that, apart from the node $\mathcal{S}_j$, none of the nodes of $\mathcal{D}_\mathcal{G}(\mathcal{S}_j)$ can belong to the cluster c_i. During the intermediate step, synchronous operations are performed at the edges of this graph in the same direction to compute each ignored value of a cluster, $ignVal(c_i)$. In the example of Figure 4.2, $F_3 \rightarrow F_2 \rightarrow x_3$ is the dependent acyclic graph for split node x_3 of cluster c_1 in the intermediate step.

As discussed, the entire operation of the intermediate step is performed by the corresponding cluster head for each cluster. Therefore, apart from receiving the M_r values, which is literally a single message from the participating clusters of the first round, there is no communication cost in this step. This produces a significant reduction of communication cost (time) in $\mathcal{PMP}$. Moreover, we can avoid the computation of variable-to-factor messages in the intermediate step as they are redundant and produce no further significance in this step. In the example of Figure 4.3, we consider every possible scenario while computing the message $F_8 \rightarrow x_0$ (i.e. $F_8 \rightarrow F_7$), and show that the variable-to-factor messages ($x_4 \rightarrow F_8, x_1 \rightarrow F_8, x_2 \rightarrow F_8, x_3 \rightarrow F_8$) are redundant during the intermediate step of $\mathcal{PMP}$ (Equations 4.11, 4.12, 4.13). Here, $Q_{1\rightarrow 8}(x_1) = \{0, 0, ..., 0\}$ and $Q_{4\rightarrow 8}(x_4)) = \{0, 0, ..., 0\}$ as x_1 and x_4 do not have any neighbours apart from F_8. As a result, we get Equation 4.13 from Equations 4.11 and 4.12, and Equation 4.14 is the generalization of Equation 4.13.

$$\begin{aligned} R_{F_8\rightarrow x_0}(x_0) &= Q_{x_0\rightarrow F_7}(x_0) \\ &= D_{F_8\rightarrow F_7}(x_0) \end{aligned} \tag{4.11}$$

$$\begin{aligned} R_{F_8\rightarrow x_0}(x_0) = \max_{\{x_0,x_1,x_2,x_3,x_4\}\setminus\{x_0\}} &[F_8(x_0,x_1,x_2,x_3,x_4) + Q_{x_1\rightarrow F_8}(x_1) + Q_{x_2\rightarrow F_8}(x_2) \\ &+ Q_{x_3\rightarrow F_8}(x_3) + Q_{x_4\rightarrow F_8}(x_4)] \\ = \max_{\{x_1,x_2,x_3,x_4\}} &[F_8(x_0,x_1,x_2,x_3,x_4) + \{Q_{x_1\rightarrow F_8}(x_1) + Q_{x_4\rightarrow F_8}(x_4)\} + \{R_{F_9\rightarrow x_2}(x_2)\} \\ &+ \{R_{F_3\rightarrow x_3}(x_3) + R_{F_4\rightarrow x_3}(x_3)\}] \\ = \max_{\{x_1,x_2,x_3,x_4\}} &[F_8(x_0,x_1,x_2,x_3,x_4) + \{R_{F_9\rightarrow x_2}(x_2) + R_{F_3\rightarrow x_3}(x_3) + R_{F_4\rightarrow x_3}(x_3)\}] \\ = \max_{\{x_1,x_2,x_3,x_4\}} &[F_8(x_0,x_1,x_2,x_3,x_4) + \{D_{F_9\rightarrow F_8}(x_2) + D_{F_3\rightarrow F_8}(x_3) + D_{F_4\rightarrow F_8}(x_3)\}] \end{aligned} \tag{4.12}$$

$$D_{F_8\rightarrow F_7}(x_0) = \max_{\{x_1,x_2,x_3,x_4\}} [F_8(x_0,x_1,x_2,x_3,x_4) + \{D_{F_9\rightarrow F_8}(x_2) + D_{F_3\rightarrow F_8}(x_3) + D_{F_4\rightarrow F_8}(x_3)\}] \tag{4.13}$$

$$D_{F_j\rightarrow F_p}(x_i) = \max_{\mathbf{x}_j\setminus x_i} [F_j(\mathbf{x}_j) + \sum_{k\in C_j\setminus F_p} D_{F_k\rightarrow F_j}(x_t)] \tag{4.14}$$

$$\mathbf{c_1}\begin{cases}\textbf{Before First Round of Message Passing}: \\ split\ node\ of\ c_1,\ \mathcal{S}=\{\mathcal{S}_1=x_3\} \\ initial\ values\ for\ x_3=\{0,0\} \\ \textbf{Before Intermediate Step}: \\ ignored\ values\ for\ x_3=\{R_{F_2\to x_3}(x_3)\} \\ \textbf{After Intermediate Step}: \\ initial\ values\ for\ x_3=\mathcal{S}_1.values=\{65,72\}\end{cases} \tag{4.15}$$

$$\mathbf{c_2}\begin{cases}\textbf{Before First Round of Message Passing}: \\ split\ node\ of\ c_2,\ \mathcal{S}=\{\mathcal{S}_1=x_3,\mathcal{S}_2=x_5\} \\ initial\ values\ for\ x_3=\{0,0\} \\ initial\ values\ for\ x_5=\{0,0\} \\ \textbf{Before Intermediate Step}: \\ ignored\ values\ for\ x_3=\{R_{F_1\to x_3}(x_3)\} \\ ignored\ values\ for\ x_5=\{R_{F_4\to x_5}(x_5)\} \\ \textbf{After Intermediate Step}: \\ initial\ values\ for\ x_3=\mathcal{S}_1.values=\{27,28\} \\ initial\ values\ for\ x_5=\mathcal{S}_2.values=\{65,72\}\end{cases} \tag{4.16}$$

$$\mathbf{c_3}\begin{cases}\textbf{Before First Round of Message Passing}: \\ split\ node\ of\ c_3,\ \mathcal{S}=\{\mathcal{S}_1=x_5\} \\ initial\ values\ for\ x_5=\{0,0\} \\ \textbf{Before Intermediate Step}: \\ ignored\ values\ for\ x_5=\{R_{F_3\to x_5}(x_5)\} \\ \textbf{After Intermediate Step}: \\ initial\ values\ for\ x_5=\mathcal{S}_1.values=\{81,82\}\end{cases} \tag{4.17}$$

Despite the aforementioned advantages, each synchronous execution (i.e. $D_{F_j\to F_p}(x_i)$) within $\mathcal{D}_{\mathcal{G}}(\mathcal{S}_j)$ is still as expensive as a factor-to-variable message, and can be computed using Equation 4.14. In this context, C_j denotes the set of indexes of the functions connected to function F_j in the dependent acyclic graph ($\mathcal{D}_{\mathcal{G}}(\mathcal{S}_j)$) of the intermediate step, and x_t stands for a variable connected to both functions F_k and F_j. Notably, Equation 4.14 retains similar properties as Equation 4.2, but the receiving node is a function node (or the split node) instead of only a variable node. For example, x_3 is a split node for cluster c_1 in Figure 4.2, where F_2 and F_3 are the nodes of the directed acyclic graph for x_3. Here, the cluster head of c_1 receives the M_r value {65, 72}. Then the first operation on the graph produces {102, 99} for the edge $F_3 \to F_2$. Subsequently, by taking {102, 99} as the input, the cluster head of c_1 generates $\{119, 126\}$, which is

Algorithm 7: Domain pruning to compute $D_{F_j \to F_p}(x_i)$ in intermediate step of $\mathcal{PMP}$

Input: Local utility of factor $F_j(\mathbf{x}_j)$: sorted independently by each state of the domain of $\mathbf{x}_j$; Incoming messages from the neighbour(s) of F_j other than F_p, where $\mathbf{n}$ is the number of neighbour(s) of F_j.

Output: Pruned ranges of values of the states over which maximization will be computed.

1 *Let* $\{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ be the states of the domain
2 $m \leftarrow \sum_{k=1}^{\mathbf{n}-1} \max(m_k)$, where m_k is one of the $\mathbf{n}-1$ messages received by F_j
3 **for** $i \leftarrow 1$ ***to*** **r do** `// for each state of the domain`
4 $p \leftarrow \max_{\mathbf{s_i}}(F_j(\mathbf{x}_j))$
5 $b \leftarrow \sum_{k=1}^{\mathbf{n}-1} val_p(m_k)$
6 $t \leftarrow m - b$
7 $j \leftarrow 1$
8 $\mathbf{q} \leftarrow getVal(j)$ `// pick a value from` $\mathbf{s_i}$ `less than` p
9 **if** $t \leq p - \mathbf{q}$ **then**
10 **result** $valueRange_{\mathbf{s_i}}[p, \mathbf{q})$
11 **else**
12 $j \leftarrow j + 1$
13 **go to** line 8

the ignored value for $x_3 \in c_1$ generated during the intermediate step. At this point, instead of $\{0, 0\}$ the second round uses $\{119, 126\}$ as the initial value for node x_3 in cluster c_1. On the other hand, cluster c_2 has two neighbouring clusters, c_1 and c_3, and neither of them have other clusters to depend on. Therefore, there is no need for further computation in the intermediate step for the split nodes x_3 and x_5 of cluster c_2. The M_r values {27, 28} and {65, 72} need to be used as the ignored values for the split node x_3 and x_5 respectively for the cluster c_2. During different steps of $\mathcal{PMP}$, all the values related to the split nodes of the clusters c_1, c_2 and c_3 are shown in the set of Equations 4.15, 4.16 and 4.17, respectively.

Note that each synchronous operation (i.e. Equation 4.14) on each edge of $\mathcal{D}_{\mathcal{G}}(\mathcal{S}_j))$ in the intermediate step still requires a significant amount of computation due to the potentially large parameter domain size and constraints with high arity. Considering this, in order to improve the computational efficiency of this step, we propose an algorithm to reduce the domain size over which the maximization needs to be computed (Algorithm 7)[3]. In other

[3]Algorithm 7 is inspired by Algorithm 2, and Algorithm 2 can also be tailored for this context.

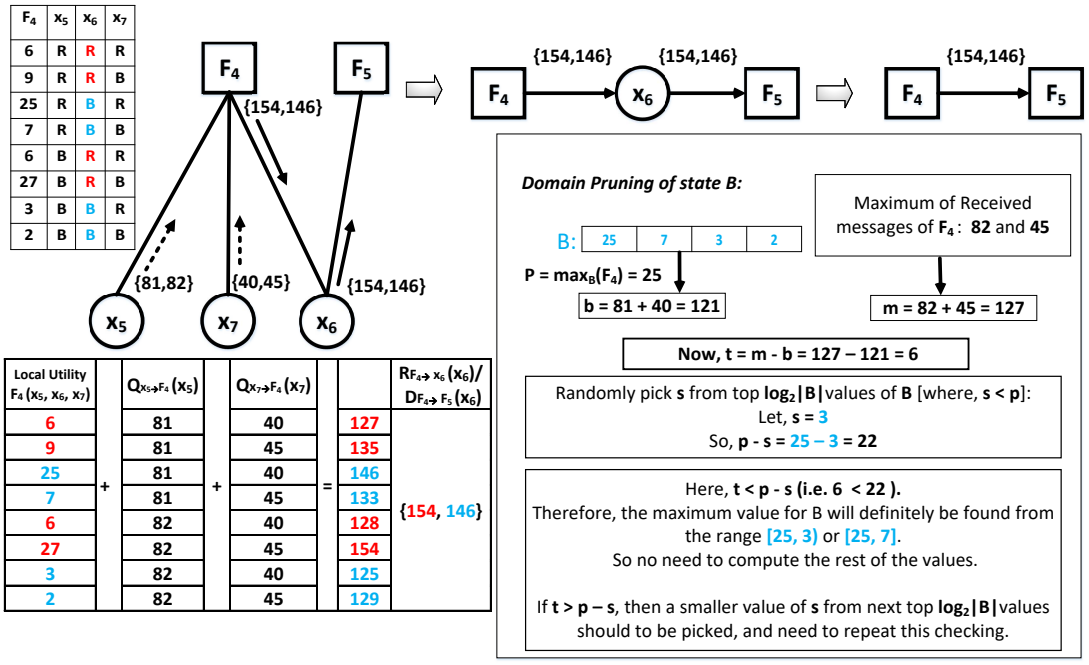


Figure 4.4: Worked example of domain pruning during the intermediate step of $\mathcal{PMP}$. In this example, red and blue colours are used to distinguish the domain state R and B while performing the domain pruning.

words, Algorithm 7 operates on Equation 4.14, that represents a synchronous operation of the intermediate step, to reduce its computational cost. This algorithm requires incoming messages from the neighbour(s) of a function in $\mathcal{D}_{\mathcal{G}}(\mathcal{S}_j)$, and each local utility must be sorted independently by each *state* of the domain. Specifically, this sorting can be done before computing the *StatusMessage* during the time of the first round of message passing. Therefore, it does not incur any further delay. Finally, this algorithm returns a pruned range of values for each state of the domain (i.e. $\{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$) over which the maximization needs to be computed.

As discussed, $D_{F_j \to F_p}(x_i)$ stands for a synchronous operation where F_j computes a value for F_p within $\mathcal{D}_{\mathcal{G}}(\mathcal{S}_j)$. Initially, line 2 computes m, which is the summation of the maximum values of the messages received by the sending function F_j, other than F_p. In the worked example of Figure 4.4, we illustrate the complete process of domain pruning for the state B, while computing a sample message from F_4 to F_5. Notably, this is the same example we previously used in Section 4.1 to explain the function-to-variable message computation process (see Equations 4.5−4.8), and it can be seen that the synchronous operation (i.e. F_4 to F_5) in the intermediate step is similar to that of the function-to-variable (F_4 to x_6) computation. Here, the messages received by the sending node F_4 are $\{81, 82\}$ and $\{40, 45\}$. As the maximum of the received messages are 82 and 45, the value of $m = 82 + 45 = 127$. Now, the *for* loop in lines $3-13$ generates the range of the values for each *state* $\mathbf{s_i} \in \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ of the domain from where we will always find the maximum value for the function F_j, and discard the rest. To do so, line 4 of the algorithm initially generates the maximum value p for the state $\mathbf{s}_i$ of the function F_j (i.e.

$\max_{\mathbf{s_i}}(F_j(\mathbf{x}_j)))$. Then, line 5 computes b, which is the summation of the corresponding values of p from the incoming messages of F_j. In the example of Figure 4.4, the sorted local utility for B is $\{25, 7, 3, 2\}$, from where we get $b = 81 + 40 = 121$ for the maximum value, $p = 25$. Subsequently, line 6 gets the base case t, which is a subtraction of b from m (i.e. $t = m - b = 127 - 121 = 6$).

At this point, a value $\mathbf{q}$, which is less than p, is picked from the sorted list of that state $\mathbf{s_i}$ (line 8). Here, the function $getVal(j)$ finds the value of $\mathbf{q}$, with j representing the number of attempts. In the first attempt (i.e. $j = 1$), it will randomly pick a value of $\mathbf{q}$ from a range of top $\log_2 |\mathbf{s_i}|$ values of $\mathbf{s_i}$, where $|\mathbf{s_i}|$ is the size of $\mathbf{s_i}$. Finally, if the value of t is less than or equal to $p - \mathbf{q}$, the desired maximization must be found within the range of $[p, \mathbf{q})$. Otherwise, we need to pick a smaller value of $\mathbf{q}$ from the next top $\log_2 |\mathbf{s_i}|$ values, and repeat the checking (lines $9 - 12$). In the worked example, we pick the value $\mathbf{q} = 3$ which is in the top 2 (i.e. $\log_2 |\mathbf{4}|$) values of B. Here, t is smaller than $p - \mathbf{q}$, that is $6 < (25 - 3)$, and it satisfies the condition of line 9. As a result, the maximum value for the *state* B will definitely be found from range $[25, 3)$ or $[25, 7]$. Hence, it is not required to consider the smaller values of $\mathbf{q}$ for this particular scenario. Eventually, introducing the domain pruning technique allows $\mathcal{PMP}$ to ignore these redundant operations during the intermediate step, thus reducing the computational cost in terms of completion time. Even for such a small example, this approach reduces half of the search space. Therefore, the overall completion time of the intermediate step can be shortened significantly by incorporating the domain pruning algorithm into it.

As mentioned earlier, $\mathcal{PMP}$ uses only the cluster heads to complete the operation of the intermediate step, instead of using all cooperating agents in the system. This phenomenon means $\mathcal{PMP}$ is a partially decentralized approach. To be exact, our approach is mostly decentralized, and the specific part (i.e. intermediate step) of the algorithm which needs to be done by the cluster heads is known in advance. Therefore, no effort is required to find these parts of a DCOP. Consequently, there is no ambiguity in deciding which part of a problem should be done by which agent, nor is there any possibility of duplicating efforts in solving overlapping problems during this step of $\mathcal{PMP}$. These are the major weaknesses that have previously been observed in deploying partial centralized techniques (see Sections 2.3.1 and 2.3.3 for more details). Thus, we effectively take the advantages of partial centralization without being affected by its major shortcomings.

4.2.4 Comparative Example

Having discussed each individual step of $\mathcal{PMP}$ separately, Figure 4.5 illustrates a complete worked example that compares the performance of SMP and $\mathcal{PMP}$ in terms of completion time. In so doing, we use the same factor graph shown in Figure 4.1. Additionally, the message computation and transmission costs for the nodes, based on which

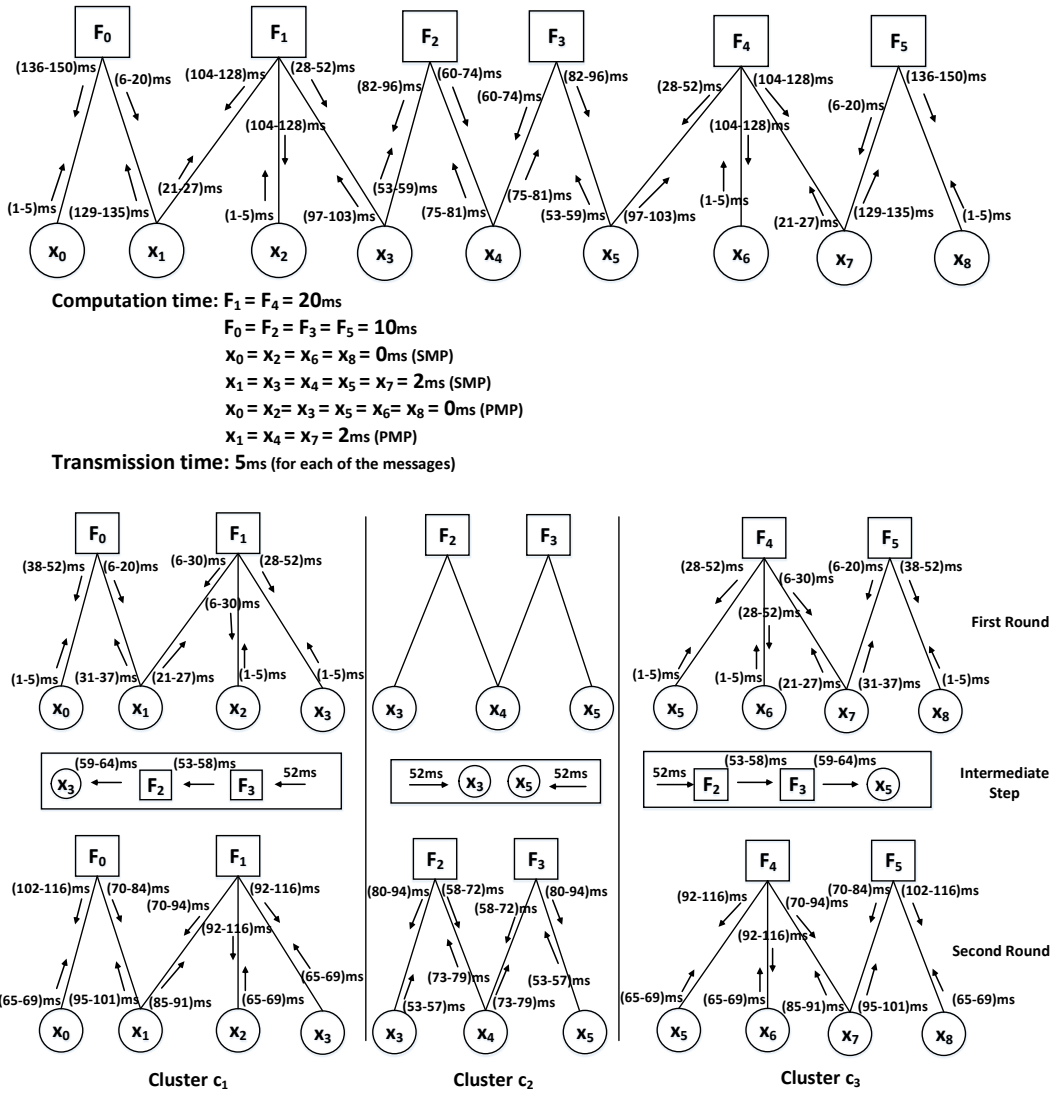


Figure 4.5: Comparative example of SMP (top) and $\mathcal{PMP}$ (bottom), in terms of completion time, based on the factor graph shown in Figure 4.1. In the figure, each edge weight within a first parentheses represents the time required to compute and transmit a message from a node to its corresponding neighbouring node. For instance, the edge weight from F_0 to x_0 in SMP is $(136-150)$ms. That means, F_0 starts computing a message for x_0 after 135ms of initiating the message passing process, and the receiving node x_0 receives the message after 150ms.

the completion time is generated, are given in the figure. In general, a function-to-variable message is computationally significantly more expensive to generate as opposed to a variable-to-function message, and a node with a higher degree requires more time to compute a message than a node with a lower degree (Kschischang et al., 2001; Lesser & Corkill, 2014; Farinelli et al., 2013). In this example, the values were chosen to reflect this observation. For instance, a function node F_1 with degree 3 requires 20ms to compute a message for any of its neighbouring nodes, and F_2 (with degree 2) requires 10ms to compute a message. On the other hand, for the variable-to-function

messages, when a variable has only one neighbouring node (e.g. x_0, x_2), it generally sends a pre-defined initial message to initiate the message passing process. Therefore, the time required to generate such a message is negligible. On the contrary, we consider 2ms as the time it takes to produce a variable-to-function message when the variable has degree 2 (e.g. x_1, x_4). Moreover, we consider 5ms as the time it takes to transmit a message from one node to another in this example. Furthermore, each edge weight within a first parentheses represents the time required to compute and transmit a message from a node to its corresponding neighbouring node. For instance, the edge weight from F_0 to x_0 in SMP is $(136 - 150)ms$. That means F_0 starts computing a message for x_0 after 135ms of initiating the message passing process, and the receiving node x_0 receives the message after 150ms.

The total calculation of the completion time following the SMP protocol is depicted at the top of the figure. At the beginning, nodes x_0, x_2, x_6 and x_8 initiate the message passing process, and their corresponding receiving nodes F_0, F_1, F_4 and F_5 receive messages after 5ms. Then, x_1 and x_7 receive messages from F_0 and F_5 respectively after 20ms. Although F_1 has already received a message from x_2, it cannot generate a message as it requires at least two messages to produce one. In this process, the message passing process will complete when all of the nodes receive messages from all of their neighbours. In this particular example, this is when x_0 and x_8 receive messages from F_0 and F_5 respectively after 150ms. Thus, the completion time of SMP is 150ms.

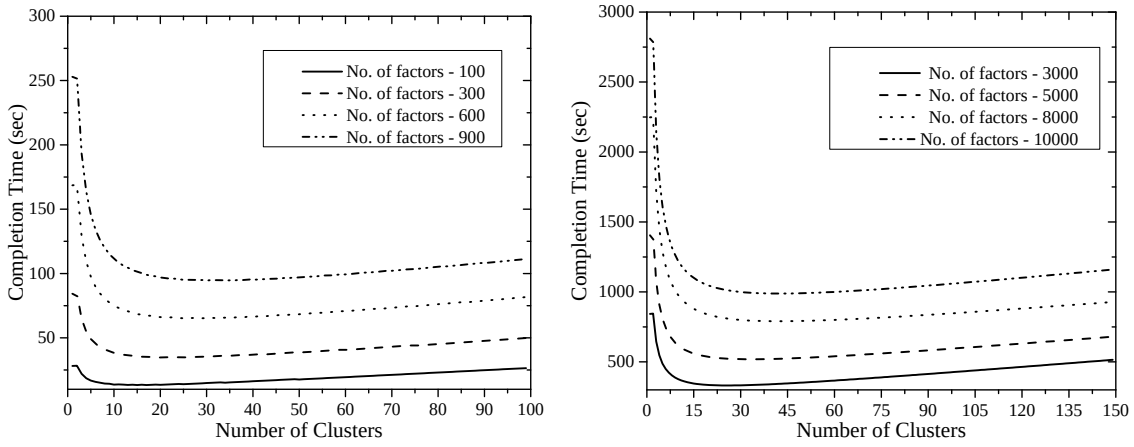
On the other hand, $\mathcal{PMP}$ splits the original factor graph into three clusters in this example. Each of the clusters executes the message passing in parallel following the similar means and regulation as its SMP counterpart. To be precise, the first round of message passing is completed after 52ms. Subsequently, the intermediate step to recover the ignored values for the split nodes is initiated. For cluster c_1, x_3 is the split node that ignored the message coming from F_2 during the first round. Two synchronous operations, $D_{F_3 \to F_2}(x_4)$ and $D_{F_2 \to x_3}(x_3)$, are required to obtain the desired value for the split node x_3. Each of these operations is as expensive as the corresponding function-to-variable messages. However, Algorithm 7 can be used to reduce the cost of these operations, and we consider a reduction of 40%, since this is the minimum reduction we get from the empirical evaluation (see Section 4.3). In this process, the intermediate step of cluster c_1 and c_3 is completed after 64ms. Unlike those two clusters, cluster c_2 shares split node x_3 (x_5) with such a cluster c_1 (c_3) that has no other cluster to depend on apart from c_2. Therefore, the ignored values for x_3 and x_5 are ready immediately after the completion of the first round (see Algorithm 6). As a result, cluster c_2 can start its second round after 52ms. In any case, the second round utilizes the recovered ignored values as the initial values for the split nodes to produce the same outcome as its SMP counterpart. We can observe that the second round of message passing completes after 116ms. Thus, even for such a small factor graph of 6 function nodes and 9 variable nodes, we can save around 23% of the completion time by replacing SMP with $\mathcal{PMP}$.

4.3 Empirical Evaluation

Given the detailed description in previous section, we now evaluate the performance of $\mathcal{PMP}$ to show how effective it is in terms of completion time compared to the benchmarking SMP protocol. To maintain the distributed nature, all the experiments were performed on a simulator in which we generated different instances of factor graph representations of DCOPs that have varying numbers of function nodes $100 - 10,000$. Hence, the completion time that is reported in this section is a simulated distributed metric, and the factor graphs are generated by randomly connecting a number of variable nodes per function from the range $2 - 7$. Although we use these ranges to generate factor graphs for this experiment, the results are comparable for larger settings. Now, to evaluate the performance of $\mathcal{PMP}$ on different numbers of clusters, we report the result for the number of clusters $2 - 99$ for the factor graph of $100 - 900$ function nodes, and $2 - 149$ for the rest. These ranges were chosen because the best performances are invariably found within these ranges, and the performance steadily gets worse for larger numbers of clusters. Since both SMP and $\mathcal{PMP}$ are generic protocols that can be applied to any GDL-based DCOP formulation, we run our experiments on the generic factor graph representations that are not limited to any particular application domain. Moreoover, instead of concentrating on the overall algorithm of a DCOP solution approach, we only focus our objective on evaluating the performance of $\mathcal{PMP}$ as opposed to SMP in terms of the completion time of the message passing process. Notably, the completion time of such algorithms mainly depends on following three parameters:

- Average time to compute a potentially expensive function-to-variable message within a factor graph, denoted as T_{p_1}.
- Average time to compute an inexpensive variable-to-function message within a factor graph, denoted as T_{p_2}.
- Average time to transmit a message between nodes of a factor graph, denoted as T_{cm}.

In the MAS literature, a number of extensions of the Max-Sum/BMS algorithms have been developed. Significantly, each of them can be defined by different ratios of the above mentioned parameters. For example, the value of $\frac{T_{p_1}}{T_{p_2}}$ is close to 1 for algorithms such as FMS, BFMS or BnB-FMS, because they restrict the domain sizes for the variables always to 2 (Ramchurn et al., 2010; Macarthur et al., 2011). In contrast, in a DCOP setting with large domain size, the value of $\frac{T_{p_1}}{T_{p_2}}$ is much higher for a particular application of the Max-Sum or the BMS algorithm (Farinelli et al., 2008; Rogers et al., 2011). Additionally, the communication cost or the average message transmission cost (T_{cm}) can vary due to different reasons such as environmental hazard in disaster response or climate monitoring application domains (Stranders et al., 2009; Vinyals et al., 2011). To reflect all these



(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.6: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E1: ($T_{p_1} > T_{p_2}$ AND $T_{p_1} \approx T_{cm}$).

issues in evaluating the performance of $\mathcal{PMP}$, we consider different ratios of those parameters to show the effectiveness of $\mathcal{PMP}$ over its SMP counterpart in a wide range of conceivable settings. To be precise, we run our experiments on seven different settings, each of which has identical ratios of the parameters: T_{p_1}, T_{p_2} and T_{cm}. Note that, once the values of each of the parameters have been fixed for a particular setting, the outcome remains unchanged for both SMP and the different versions of $\mathcal{PMP}$ even if we repeat the experiments for that setting. This is because we run both the protocols on acyclic or transformed acyclic version of a factor graph, hence they always provide a deterministic outcome. Hence, there is no need to perform an analysis of statistical significance for this set of experiments. Note that all of the following experiments are performed on a simulator implemented on an Intel i7 Quadcore 3.4GHz machine with 16GB of RAM.

4.3.1 Experiment E1: $T_{p_1} > T_{p_2}$ AND $T_{p_1} \approx T_{cm}$

Figures 4.6(a) and 4.6(b) illustrate the comparative measure on completion time for SMP and $\mathcal{PMP}$ under experimental setting E1 for the factor graph with the number of function nodes $100 - 900$ and $3000 - 10,000$, respectively. Each line of the figures shows the result of both SMP (Number of Clusters $= 1$) and $\mathcal{PMP}$ (Number of Clusters > 1). The setting E1 characterizes a scenario where average computation cost (time) of a function-to-variable message (T_{p_1}) is moderately more expensive than a variable-to-function message (T_{p_2}), and the average time to transmit a message between nodes (T_{cm}) is approximately similar to T_{p_1}. To be precise, we consider T_{p_2} be 100 times less expensive than a randomly taken T_{p_1} for this particular experiment. The scenario E1 is commonly seen in the following GDL-based DCOP algorithms: Max-Sum, BMS and FMS. Once these three parameters have been determined, the completion time of SMP (i.e. T_{smp}) and $\mathcal{PMP}$ (i.e. T_{pmp}) can be generated using Equation 4.18 and

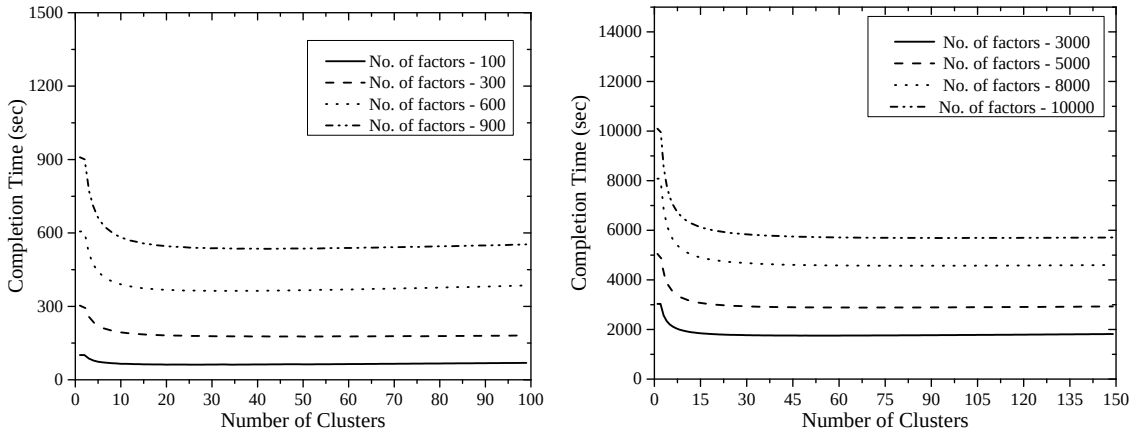
Equation 4.19, respectively. Here, the function $requiredTime()$ takes T_{p_1}, T_{p_2}, T_{cm} and an acyclic factor graph F_G as inputs, and computes the time it needs to finish the message passing by following the regulation of SMP.

$$T_{smp} = requiredTime(F_G, T_{p_1}, T_{p_2}, T_{cm}) \tag{4.18}$$

$$T_{pmp} = 2 \times requiredTime(c_{largest}, T_{p_1}, T_{p_2}, T_{cm}) + T_{intm} \tag{4.19}$$

As discussed in Section 4.2, due to parallel execution on each cluster, for $\mathcal{PMP}$, we only need to consider the largest cluster of F_G (i.e. $c_{largest}$) instead of the complete factor graph F_G. Altogether the completion time of $\mathcal{PMP}$ includes the time required to complete the two rounds of message passing on the largest cluster with the addition of the time it takes to complete the intermediate step (T_{intm}). In the intermediate step, each synchronous operation is as expensive as the factor-to-variable message (T_{p_1}). However, during the intermediate step, the proposed domain pruning technique of $\mathcal{PMP}$ (i.e. Algorithm 7) minimizes the cost of T_{p_1} by reducing the size of the domain (i.e search space) over which maximization needs to be computed. To empirically evaluate the performance of the domain pruning technique, we independently test it on a randomly generated local utility table that has a varying domain size from 2 to 20. In general, we observe a significant reduction of the search space, ranging from around 35% to 75%, by using this technique, and as expected the results are getting better with the increase of the domain size (see Section 4.2.3). Hence, to reflect the worst case scenario, we consider only a 35% reduction for each operation of the intermediate step while computing the completion time of $\mathcal{PMP}$ for all the results reported in this experiment.

According to Figure 4.6(a), the best performance of $\mathcal{PMP}$ compared to the SMP protocol can be found if the number of clusters is picked from the range $\{5 - 25\}$. In particular, for the smaller factor graphs this range becomes smaller. For example, when we are dealing with a factor graph of 100 function nodes the best results are found within the range of $\{5-18\}$ clusters; afterwards, the performance of $\mathcal{PMP}$ gradually decreases. This is because the time required to complete the intermediate step increases steadily when the cluster size gets smaller (i.e. the number of clusters gets larger). On the other hand, the time it takes to complete the two rounds of message passing increases when the cluster size becomes larger. As a consequence, it is observed from the results that the performance of $\mathcal{PMP}$ drops steadily with the increase of the number of clusters after reaching to its peak with a certain number of clusters. Generally, we observe a similar trend in each scenario. Therefore, a proper balance is necessary to obtain the best possible performance from $\mathcal{PMP}$ (see Section 4.4). Notably, for the larger factor



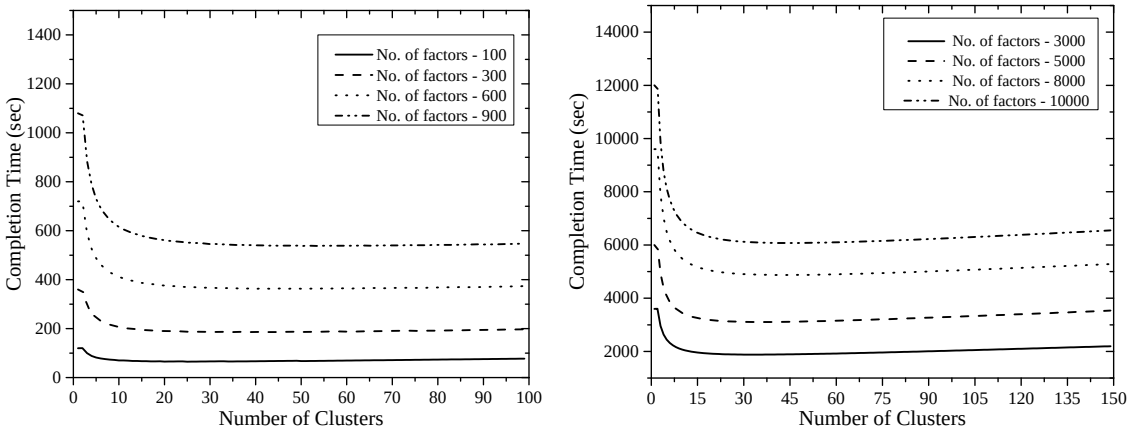
(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.7: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E2: ($T_{p_1} \gg T_{p_2}$ AND $T_{p_1} \gg T_{cm}$).

graphs, the comparative performance gain of $\mathcal{PMP}$ is more substantial in terms of completion time due to the consequence of parallelism. As observed, $\mathcal{PMP}$ running over a factor graph with $100 - 300$ function nodes achieves around 53% to 59% performance gain (Figure 4.6(a)) over its SMP counterpart. On the other hand, $\mathcal{PMP}$ takes 61% to 63% less time than SMP when larger factor graphs ($600 - 900$ functions) are considered. Finally, Figure 4.6(b) depicts that this performance gain reaches around 61% to 65% for the factor graph having 3000 to 10,000 function nodes. Here, this performance gain of $\mathcal{PMP}$ is achieved when the number of clusters is chosen from the range of $\{25 - 44\}$.

4.3.2 Experiment E2: $T_{p_1} \gg T_{p_2}$ AND $T_{p_1} \gg T_{cm}$

In experimental setting E2, we generated the results based on similar comparative measures and representations as the setting E1 (Figure 4.7). However, E2 characterizes the scenario where the average computation cost (time) of a function-to-variable message (T_{p_1}) is extremely expensive compared to the variable-to-function message (T_{p_2}), and the average time to transmit a message between nodes (T_{cm}) is considerably more inexpensive as opposed to T_{p_1}. To be exact, we consider T_{p_2} be 10,000 times less expensive than a randomly taken T_{p_1} for this particular setting. Here, T_{p_1} is considered 200 times more time consuming than T_{cm}. Max-Sum and BMS are two exemplary GDL-based algorithms where E2 is commonly seen. More specifically, this particular setting reflects those applications that contain variables with high domain size. For example, assume the domain size is 15 for all 5 variables associated with a function. In this case, to generate each of the function-to-variable messages, the corresponding agent needs to perform 15^5 or $7,59,375$ operations. Since T_{p_1} is extremely expensive in this experimental setting, the performance of $\mathcal{PMP}$ largely depends on the performance of the domain



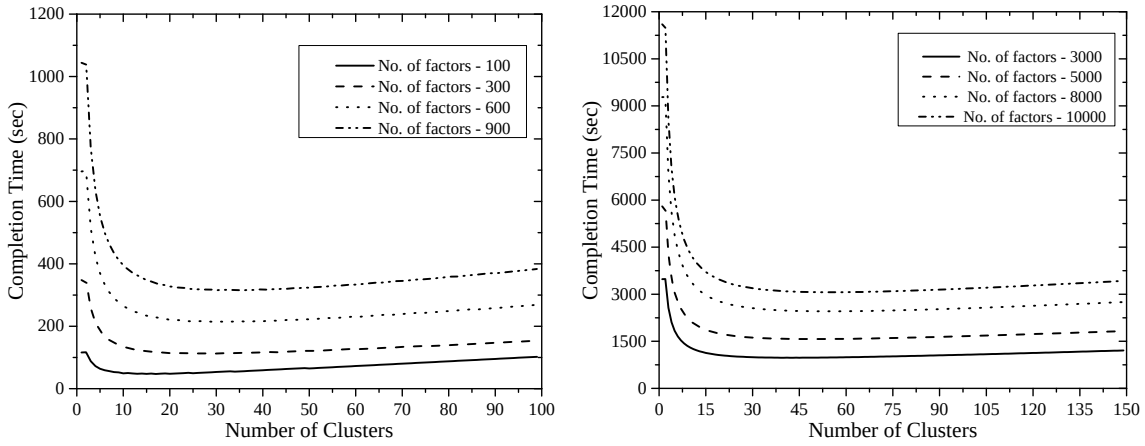
(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.8: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E3: ($T_{p_1} \gg T_{p_2}$ AND $T_{p_1} > T_{cm}$).

pruning technique. Similar to the above experiment, Figure 4.7(a) shows the results for the factor graphs having $100 - 900$ function nodes, and the results obtained by applying on larger factor graphs ($3000 - 10000$ function nodes) are shown in Figure 4.7(b). This time, the best performance of $\mathcal{PMP}$ for those two cases are observed when the number of clusters are picked from the ranges $\{15 - 41\}$ and $\{45 - 55\}$, respectively. Afterwards, the performance of $\mathcal{PMP}$ drops gradually due to the same reason as observed in E1. Notably, the performance gain reaches around 37% to 42% for the factor graphs having $100 - 900$ function nodes, and 41% to 43% for $3000 - 10000$ function nodes.

4.3.3 Experiment E3: $T_{p_1} \gg T_{p_2}$ AND $T_{p_1} > T_{cm}$

Experimental setting E3 possesses similar properties and scenarios as E2, apart from the fact that here T_{p_1} is moderately more expensive than T_{cm} instead of extremely more expensive. Similar to the previous experiment, we consider T_{p_2} be 10000 times less expensive than a randomly taken T_{p_1}. However, T_{cm} is taken only 10 times less expensive than T_{p_1}. It is observed from the results that even without the domain pruning technique, $\mathcal{PMP}$ minimizes the cost of T_{cm} and T_{p_2} significantly in E3. This is because T_{cm} is not too inexpensive and the operations of the intermediate step do not include any communication cost. Moreover, given T_{p_1} is also very expensive, $\mathcal{PMP}$ produces better performance than what we observed in E2 by utilizing the domain pruning technique. Altogether, Figures 4.8(a) and 4.8(b) show that $\mathcal{PMP}$ consumes 45% to 49% less time than SMP for this setting when the number of clusters is chosen from the range $\{17 - 47\}$. Max-Sum and BMS are the exemplary algorithms where settings similar to E3 are commonly seen.



(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

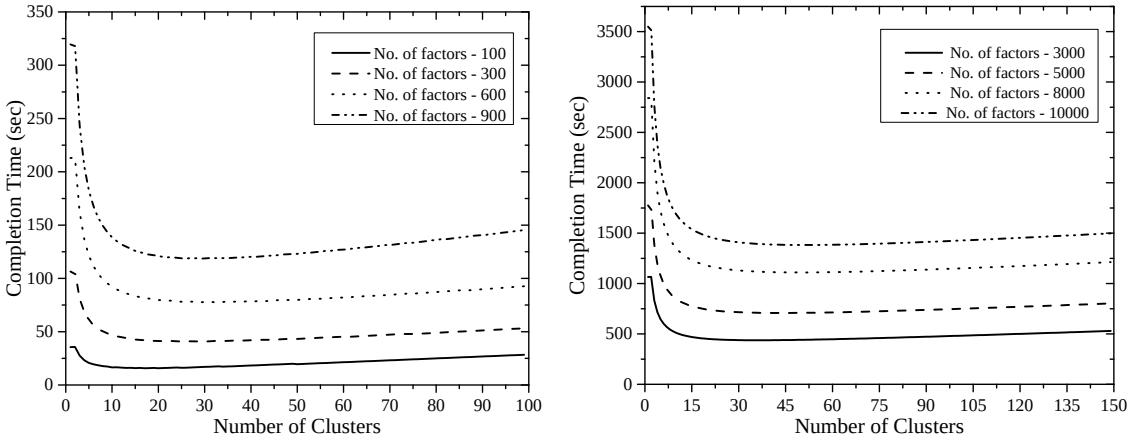
Figure 4.9: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E4: ($T_{p_1} \gg T_{p_2}$ AND $T_{p_1} \approx T_{cm}$).

4.3.4 Experiment E4: $T_{p_1} \gg T_{p_2}$ AND $T_{p_1} \approx T_{cm}$

Figure 4.9 shows the comparative results of $\mathcal{PMP}$ over SMP for experimental setting E4. E4 characterizes the scenarios where T_{p_1} is extremely more expensive than T_{p_2}, and approximately equal to T_{cm}. To be exact, we consider T_{p_2} be approximately 5000 times less expensive than randomly taken values of T_{p_1} and T_{cm}. Here, both T_{p_1} and T_{cm} are substantial, and hence $\mathcal{PMP}$ achieves notable performance gains over SMP, compared to the previous experiments. According to the graphs of Figures 4.9(a) and 4.9(b), $\mathcal{PMP}$ takes 59% to 73% less time compared to its SMP counterpart. The preferable range of number of clusters for the setting E4 is $\{15 - 55\}$.

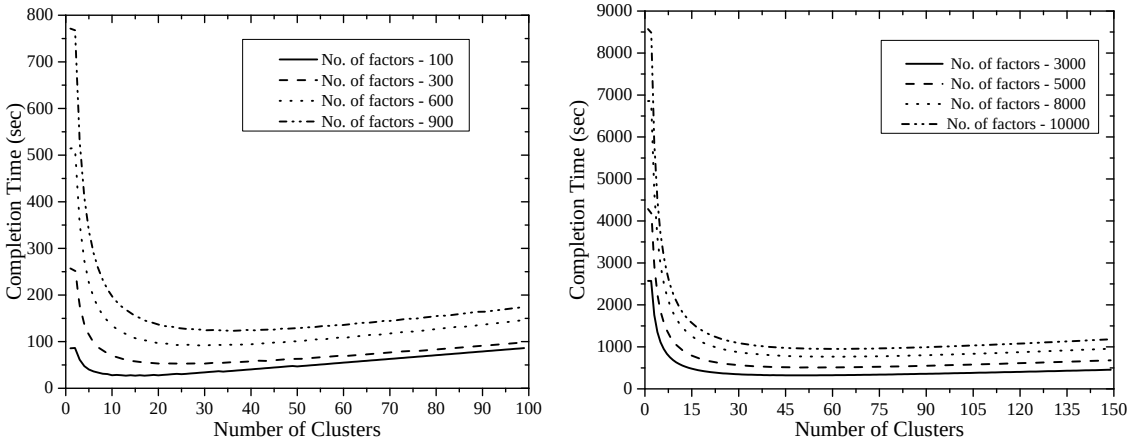
4.3.5 Experiment E5: $T_{p_1} \approx T_{p_2}$ AND $T_{p_1} \approx T_{cm}$

Experiment E5 characterizes the scenarios where T_{p_1} and T_{cm} are approximately equal to the inexpensive T_{p_2}. Such a scenario normally occurs when message passing (SMP or $\mathcal{PMP}$) is applied on the following algorithms: FMS, BnB Max-Sum, G-FBP or Max-Sum/BMS with small domain size and inexpensive communication cost in terms of time. This is a trivial setting where each of the three parameters is not that expensive. Specifically, as T_{p_1} is inexpensive, the domain pruning technique has less impact on reducing the completion time of $\mathcal{PMP}$. However, the effect of parallelism from the clustering process coupled with the avoidance of redundant variable-to-function messages during the intermediate step allows $\mathcal{PMP}$ to take 55% to 67% less time than its SMP counterpart (Figures 4.10(a) and 4.10(b)). The preferable range of number of clusters is same as the setting E4.



(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.10: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E5: ($T_{p_1} \approx T_{p_2}$ AND $T_{p_1} \approx T_{cm}$).

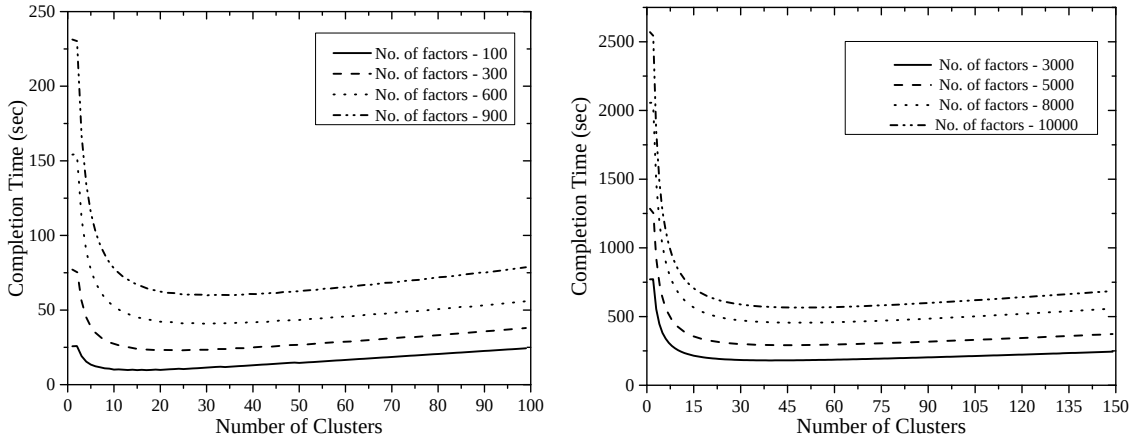


(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.11: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E6: ($T_{p_1} \approx T_{p_2}$ AND $T_{p_1} \ll T_{cm}$).

4.3.6 Experiment E6: $T_{p_1} \approx T_{p_2}$ AND $T_{p_1} \ll T_{cm}$

Figure 4.11 illustrates the comparative results of $\mathcal{PMP}$ over SMP for experimental setting E6, which possess similar properties, scenarios and the applied algorithms as E5. However, in E6 the average message transmission cost T_{cm} is considerably more expensive than T_{p_1} and T_{p_2}. To be exact, we consider T_{p_1} be 15 times less expensive than a randomly taken value of T_{cm}. As T_{cm} is markedly more expensive and T_{p_2} is approximately equal to T_{p_1}, the performance gain of $\mathcal{PMP}$ increases to the highest level (70% to 91%). To be precise, the reduction of communication by avoiding the variable-to-function messages during the intermediate step, which is extremely expensive in this setting, helps $\mathcal{PMP}$ achieves this performance. This result signifies that $\mathcal{PMP}$



(a) Number of function nodes (factors): $100 - 900$. (b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.12: Completion time: Standard Message Passing (Number of Cluster=1); Parallel Message Passing (Number of Cluster >1) for the experimental setting, E7: ($T_{p_1} \approx T_{p_2}$ AND $T_{p_1} < T_{cm}$).

performs best in those settings where the communication cost is expensive. Note that the preferable range of number of clusters for the setting E6 is $\{15 - 61\}$.

4.3.7 Experiment E7: $T_{p_1} \approx T_{p_2}$ AND $T_{p_1} < T_{cm}$

Experiment E7 possess similar properties, scenarios and exemplary algorithms as the setting E6, with the following exception. T_{cm} in E7 is moderately more expensive than T_{p_1} instead of considerably more expensive. To be precise, we consider T_{p_1} be 4 times less expensive than randomly taken values of T_{cm}. Due to the less substantial value of T_{cm}, unlike E6 where the performance gain reaches to the maximum level, $\mathcal{PMP}$ consumes 65% to 82% less time compared to its SMP counterpart (Figures 4.12(a) and 4.12(b)). The preferable range of number of clusters for E7 is $\{17 - 50\}$.

4.3.8 Total Number of Messages

The most important finding to emerge from the results of the experiments is that $\mathcal{PMP}$ significantly reduces the completion time of the GDL-based message passing algorithms for all the settings. However, $\mathcal{PMP}$ requires more messages to be exchanged compared to its SMP counterpart due to two rounds of message passing. To explore this trade-off, Figure 4.13 illustrates the comparative results of $\mathcal{PMP}$ and SMP in terms of the total number of messages for factor graphs with a number of function nodes $50 - 1200$ with an average 5 variables connected to a function node. The results are comparable for settings with higher arities. Specifically, we find that $\mathcal{PMP}$ needs $27 - 45\%$ more messages than SMP for a factor graph having less than 500 function nodes and $15 - 25\%$ more messages for a factor graph having more than 500 nodes. As more messages are

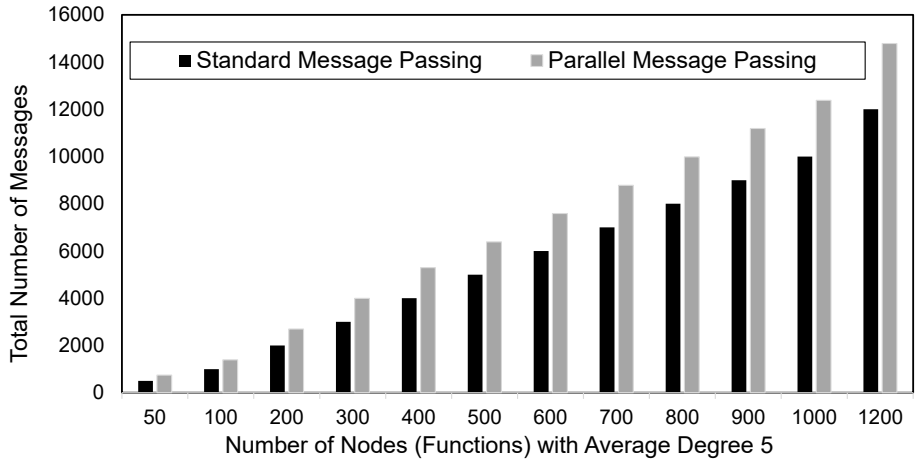


Figure 4.13: Total number of messages: SMP vs $\mathcal{PMP}$.

exchanged at the same time in $\mathcal{PMP}$ due to the parallel execution, this phenomenon does not affect the performance gain in terms of completion time.

Now, based on the extensive empirical results, we can claim that, in $\mathcal{PMP}$, even randomly splitting a factor graph into a number of clusters, within the range of around 10 to 50 clusters, always produces a significant reduction in completion time of GDL-based DCOP algorithms. However, this performance gain is neither guaranteed to be the optimal one, nor deterministic for a given DCOP setting. Therefore, we need an approach to predict how many clusters would produce the best performance from $\mathcal{PMP}$ for a given scenario. At this point, we only have a range from which we should pick the number of clusters for a certain factor graph representation of a DCOP.

4.4 Approximating the Appropriate Number of Clusters for a DCOP

In this section, we turn to the challenge of determining the appropriate number of clusters for a given scenario in $\mathcal{PMP}$. The ability to predict a specific number in this regard would allow $\mathcal{PMP}$ to split the original factor graph representation of a DCOP accurately into a certain number of clusters, prior to executing the message passing. In other words, this information allows $\mathcal{PMP}$ to be applied more precisely in different multi-agent DCOPs. However, it is not possible to predict the optimal number of clusters due to the diverse nature of the application domains, and the fact that a graphical representation of a DCOP can be altered at runtime. Therefore, we use an approximation. To be precise, we use a linear regression method, and run it off-line to approximate a specific number of clusters for a DCOP before initiating the message passing of $\mathcal{PMP}$. In this context, logistic regression, Poisson regression and a number of classification models could be used to predict information from a given finite data set.

However, they are more suited to estimate categorical information rather than predicting specific numerical data required for our model (Han et al., 2011). Therefore, we choose the linear regression method for our setting. Moreover, this method is time efficient in terms of computational cost because as an input it only requires an approximate number of function nodes of the corresponding factor graph representation of a DCOP in advance.

The remainder of this section is organised as follows. In Section 4.4.1, we explain the linear regression method, and detail of how it can be used along with the $\mathcal{PMP}$ protocol to predict the number of clusters for a specific problem instance. Then, Section 4.4.2 presents our empirical results of using this method on different experimental settings (i.e. E1, E2, ..., E7) defined and used in the previous section. Specifically, we show the differences in performance of $\mathcal{PMP}$ considering the prediction method compared to its best possible results in terms of completion time. Notably, $\mathcal{PMP}$'s performance gain, for each value within the preferred range of number of clusters, is shown in the graphs of the previous section. Here, we run a similar experiment to obtain the best possible performance gain for a certain problem instance, and then compare this with the gain obtained by using the predicted number of clusters. Finally, we end this section by evaluating the performance of $\mathcal{PMP}$ as opposed to SMP on two explicit implementations of GDL-based algorithms.

4.4.1 Determining the Appropriate Number of Clusters

Regression analysis is one of the most widely used approaches for numeric prediction (Kutner et al., 2004; Han et al., 2011). The regression method can be used to model the relationship between one or more independent or predictor variables and a dependent or response variable which is continuous valued. Many problems can be solved by linear regression, and even more can be handled by applying transformations to the variables so that a non-linear problem can be converted to a linear one. Specifically, the linear regression with a single predictor variable is known as straight-line linear regression, meaning it only involves a response variable $\mathcal{Y}$ and a single predictor variable $\mathcal{X}$. Here, the response variable $\mathcal{Y}$ is modelled as a linear function of the predictor variable $\mathcal{X}$ (Equation 4.20).

$$\mathcal{Y} = \mathcal{W}_0 + \mathcal{W}_1\mathcal{X} \tag{4.20}$$

$$\mathcal{W}_1 = \frac{\sum_{i=1}^{|\mathcal{D}|}(\mathcal{X}_i - \overline{\mathcal{X}})(\mathcal{Y}_i - \overline{\mathcal{Y}})}{\sum_{i=1}^{|\mathcal{D}|}(\mathcal{X}_i - \overline{\mathcal{X}})^2} \tag{4.21}$$

$$\mathcal{W}_0 = \overline{\mathcal{Y}} - \mathcal{W}_1\overline{\mathcal{X}} \tag{4.22}$$

Table 4.1: Sample training data from Figures 4.6 – 4.12.

Number of function nodes ($\mathcal{X}$)	Number of clusters ($\mathcal{Y}$)	Experimental setting
3000	25	E1
5000	33	E1
8000	38	E1
10000	40	E1
3000	47	E2
5000	50	E2
8000	52	E2
10000	55	E2
3000	32	E3
5000	36	E3
8000	44	E3
10000	47	E3
3000	40	E4
5000	50	E4
8000	52	E4
10000	55	E4
3000	38	E5
5000	46	E5
8000	50	E5
10000	52	E5
3000	50	E6
5000	55	E6
8000	58	E6
10000	61	E6
3000	40	E7
5000	46	E7
8000	49	E7
10000	50	E7

Table 4.2: Predicted number of clusters by applying the straight-line linear regression (Equations $4.20 - 4.22$) on the training data of Table 4.1.

Number of function nodes ($\mathcal{X}$)	Predicted number of clusters ($\mathcal{Y}$)
3050	41
4500	43
5075	44
6800	47
7500	48
8020	49
8050	49
9200	51
9975	52

In Equation 4.20, the variance of $\mathcal{Y}$ is assumed to be constant, and $\mathcal{W}_0$ and $\mathcal{W}_1$ are regression coefficients which can be thought of as weights. These coefficients can be solved by the *method of least squares*, which estimates the best-fitting straight line as the one that minimizes the error between the actual data and the estimate of the line. Let $\mathcal{D}$ be the training set consisting of values of the predictor variable $\mathcal{X}$ and their associated values for the response variable $\mathcal{Y}$. This training set contains $|\mathcal{D}|$ data points of the form $(\mathcal{X}_1, \mathcal{Y}_1), (\mathcal{X}_2, \mathcal{Y}_2), \ldots, (\mathcal{X}_{|\mathcal{D}|}, \mathcal{Y}_{|\mathcal{D}|})$. Equations 4.21 and 4.22 are used to generate the regression coefficients $\mathcal{W}_1$ and $\mathcal{W}_0$, respectively.

Now, the linear regression analysis can be used to predict the number of clusters for a certain application given that continuously updated training data from the experimental results of $\mathcal{PMP}$ exists. To this end, Table 4.1 contains the sample training data taken from the results shown in Section 4.3. Here, we formulate this training data $\mathcal{D}$ so that straight-line linear regression can be applied, where $\mathcal{D}$ consists of the values of a predictor variable $\mathcal{X}$ (number of function nodes) and their associated values for a response variable $\mathcal{Y}$ (number of clusters). In more detail, this training set contains $|\mathcal{D}|$ (number of nodes - number of clusters) data of the form $(\mathcal{X}_1, \mathcal{Y}_1), (\mathcal{X}_2, \mathcal{Y}_2), \ldots, (\mathcal{X}_{|\mathcal{D}|}, \mathcal{Y}_{|\mathcal{D}|})$. Initially, Equations 4.21 and 4.22 are used to generate regression coefficients $\mathcal{W}_1$ and $\mathcal{W}_0$ respectively, which are used to predict the appropriate number of clusters (response variable, $\mathcal{Y}$) for a factor graph with a certain number of function nodes (predictor variable, $\mathcal{X}$) (Equation 4.20). For instance, based on the training data of Table 4.1, we can predict that for factor graphs with 4500 and 9200 function nodes $\mathcal{PMP}$ should split the graphs into 43 and 51 clusters, respectively (Table 4.2). As we need to deal with only a single predictor variable, we are going to use the terms linear regression and straight-line linear

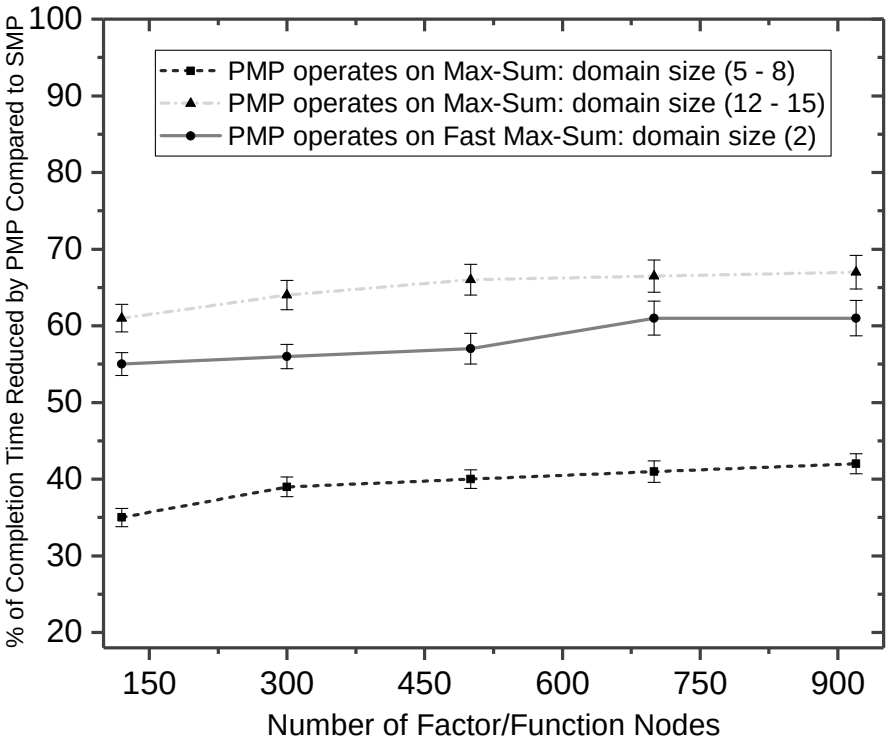
Table 4.3: Performance gain of $\mathcal{PMP}$ using the linear regression method compared to the highest possible gain from $\mathcal{PMP}$.

Number of function nodes	Experiment: E1		Experiment: E6	
	Best possible performance from $\mathcal{PMP}$	Performance of $\mathcal{PMP}$ using linear regression	Best possible performance from $\mathcal{PMP}$	Performance of $\mathcal{PMP}$ using linear regression
3050	60.69%	60.21%	88.82%	88.12%
5075	63.08%	62.50%	89.45%	89.10%
6800	63.95%	59.62%	89.80%	89.35%
8050	64.82%	59.62%	90.17%	89.80%
9975	64.83%	64.18%	90.25%	89.93%

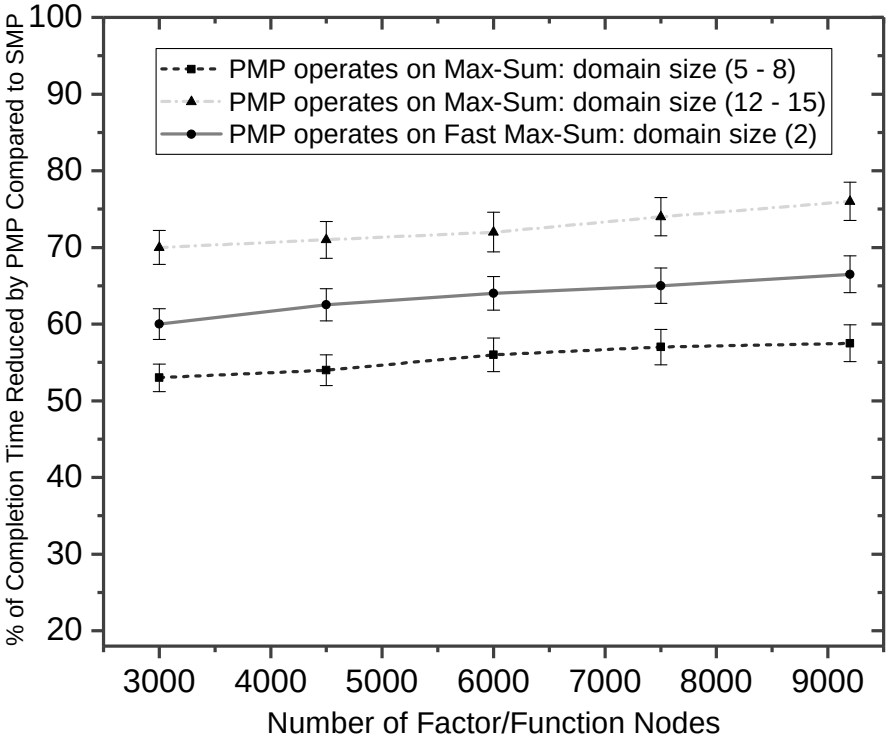
regression interchangeably. In the remainder of this section, we evaluate the performance of this extension through extensive empirical evidence.

4.4.2 Empirical Evaluation

In this section, we evaluate the performance of $\mathcal{PMP}$ by considering the number of clusters predicted using the linear regression method in terms of completion time, and compare this with the highest possible performance gain from $\mathcal{PMP}$, which is the best case outcome of $\mathcal{PMP}$ using a certain number of clusters. In so doing, we use the same experimental settings (E1, E2, ..., E7) used in Section 4.3. Specifically, Table 4.3 illustrates the comparative performance gain of $\mathcal{PMP}$ using the straight-line linear regression and the highest possible gain for five factor graphs having the number of function nodes: $3050, 5075, 6800, 8050$ and 9975 based on the experimental setting E1 and E6. We repeat the experiments of Section 4.3 for each of these factor graphs to obtain the highest possible performance gain from $\mathcal{PMP}$. That is, we reported the performance of $\mathcal{PMP}$ for all the clusters ranging from $2 - 150$. From these results, we get the highest possible gain and the performance based on the predicted number of clusters of $\mathcal{PMP}$. It can be seen that for the factor graph with 3050 function nodes the highest possible gain of $\mathcal{PMP}$ reaches to 60.69%, meaning $\mathcal{PMP}$ takes 60.69% less time to complete the message passing operation to solve the DCOP representing the factor graph than its SMP counterpart. Now, when $\mathcal{PMP}$ is applied considering the predicted number of clusters (i.e. 41) using the straight-line linear regression (Table 4.2), the gain reaches to 60.21%. This indicates, $\mathcal{PMP}$ ensures 98.7% of its possible performance gain by applying the straight-line linear regression. Similarly, $\mathcal{PMP}$ ensures 99.64%



(a) Number of function nodes (factors): $100 - 900$.



(b) Number of function nodes (factors): $3000 - 10000$.

Figure 4.14: Empirical performance of $\mathcal{PMP}$ vs SMP running on two GDL-based algorithms. Error bars are calculated using standard error of the mean.

of the possible performance gain for a factor graph with 9975 function nodes in the experimental setting E6 while applied based on the number of clusters obtained from the linear regression method. Notably, this trend is common for the rest of the factor graphs, and in each case more than 98.42% of the best possible results is assured by applying the straight-line linear regression according to the results shown in Table 4.3, and all the results are comparable for all the other experimental settings. Significantly, it can be ascertained from our experiments that a minimum of around 98.5% of the best possible results of $\mathcal{PMP}$ can be achieved when the number of clusters is predicted by the straight-line linear regression method. Notably, a common phenomenon is noticed from the empirical evaluation of Section 4.3 that the performance of $\mathcal{PMP}$ falls very slowly after it reaches to its peak by a certain number of clusters on either increasing or decreasing that number. This is why, approximating a number of clusters produces such good results.

In the final experiment, we analyse the performance of $\mathcal{PMP}$ (compared to SMP) based on two GDL-based algorithms: Max-Sum and Fast Max-Sum. We do this to observe the performance of $\mathcal{PMP}$ on the actual runtime metric that complements our controlled and systematic experiments of Section 4.3. In the experiment, we consider the predicted number of clusters obtained using the linear regression method. Here, the factor graphs are generated in the same way as they are for the experiments of Section 4.3. Additionally, we make use of the Frodo framework (Léauté et al., 2009) to generate local utility tables (i.e. cost function) for the function nodes of the factor graphs. On the one hand, we use two ranges, $(5-8)$ and $(12-15)$ of the variables' domain size to generate the utility tables for Max-Sum. In doing so, we are able to observe the comparative result for different ratios of the parameters T_{p_1} and T_{p_2}. To be precise, these two ranges reflect the scenarios $T_{p_1} > T_{p_2}$ and $T_{p_1} \gg T_{p_2}$, respectively. On the other hand, we restrict the domain size to exactly 2 for all the variables in case of Fast Max-Sum so that it reflects the characteristic of the algorithm (i.e. $T_{p_1} \approx T_{p_2}$). Notably, it is not possible to emulate a realistic application, such as disaster response or climate monitoring in a simulated environment that provides the actual value of T_{cm} (Sultanik et al., 2008). Consequently, we observe the value of T_{cm} to be very small in this experiment.

It can be seen from the solid-grey line of Figure 4.14(a) that $\mathcal{PMP}$ takes around $55-60\%$ less time than SMP to complete the message passing process for Fast Max-Sum on the factor graph having $100-900$ function nodes. Meanwhile $\mathcal{PMP}$ reduces $35-42\%$ of SMP's completion time for Max-Sum where the variables' domain size is picked from the range $5-8$ (dashed-black line). This is because in the former case, all three parameters (i.e. T_{p_1}, T_{p_2} and T_{cm}) are small and comparable. Therefore, the parallel execution of message passing, along with the avoidance of variable-to-factor messages in the intermediate step, allows $\mathcal{PMP}$ attain this performance in this case. In contrast, its performance in the latter case mainly depends on the impact of domain reduction in the

intermediate step, given that the value of T_{p_2} and T_{cm} is negligible when compared to T_{p_1}. The same holds true for Max-Sum with a larger domain size, where we observe a $67 - 72\%$ reduction in completion time by $\mathcal{PMP}$, as opposed to its SMP counterpart (dashed-grey line). However, the observed outcome in this case indicates that the impact of domain reduction in intermediate step gets better with an increase in domain size. Figure 4.14(b) illustrates a similar trend in the performance of $\mathcal{PMP}$, wherein we take larger factor graphs of 3000 to around 10000 function nodes into consideration. Here, we observe an even better performance for each of the cases, due to the impact of parallelism in larger settings.

4.5 Summary

In this chapter, we propose a generic message passing protocol which significantly reduces the completion time of GDL-based DCOP algorithms while maintaining the same solution quality. This is what we formally set as our key research challenge **C2** at the beginning of this thesis. To be precise, our approach is applicable to all the GDL-based algorithms that use factor graph as the graphical representation of DCOPs. In particular, we achieve a significant reduction in completion time for such algorithms, ranging from a reduction of $37 - 91\%$ depending on the prevailing scenario. In order to attain a performance of this quality, we introduced a cluster based method to parallelize the message passing procedure. Additionally, a domain reduction algorithm is proposed to further minimize the cost of each operation in the intermediate step. Subsequently, we addressed the challenge of determining the appropriate number of clusters for a given scenario. In doing so, we propose the use of a linear regression prediction method for approximating the appropriate number of clusters for a DCOP. Remarkably, we observe through empirical results that more than 98% of the best possible outcomes can be achieved if $\mathcal{PMP}$ is applied on the number of clusters predicted by the straight-line linear regression. In other words, if we know the size of a factor graph representing a DCOP before performing the message passing, we can utilize the straight-line linear regression method to ascertain how many clusters should be created from that factor graph. Thus, we make $\mathcal{PMP}$ a deterministic approach. Given this, by taking advantage of the $\mathcal{PMP}$ protocol, we can now use GDL-based algorithms to rapidly and effectively solve large-scale DCOPs.

Chapter 5

Speeding Up via Efficient Node-to-Agent Mapping

As discussed in Chapter 2, very little research has been carried out so far to find a good node-to-agent mapping for a DCOP. More significantly, none of them aim to reduce the completion time of DCOP algorithms through an effective mapping process. Our final research challenge **C3** has emerged in light of this backdrop, and to address this we develop a new way of speeding up GDL-based message passing algorithms that effectively solve DCOPs in multi-agent systems. In particular, we propose a new time-efficient heuristic to determine a near-optimal **M**apping of **N**odes to the participating **A**gents ($\mathcal{MNA}$) while taking cognisance of the fact that finding an optimal mapping is an NP-hard problem on its own. $\mathcal{MNA}$ is a pre-processing step that works before executing the optimization process of a DCOP algorithm. Specifically, $\mathcal{MNA}$ can be executed in a centralized or decentralized manner, based on the given application[1]. As a pre-processing step, $\mathcal{MNA}$ does not alter any internal process of the original DCOP algorithm, thereby preserving its solution quality. Additionally, the decentralized version of $\mathcal{MNA}$ specifically accommodates scenarios where the graphical representation of a DCOP experiences change(s) during the runtime of an algorithm.

The remainder of the chapter is organized as follows. In Section 5.1, we formulate the phase of node-to-agent mapping as an optimization problem, where the goal is to find an assignment that minimizes the completion time of a GDL-based exact/non-exact DCOP algorithm that operates on this mapping. In Section 5.2, we discuss the details of both the centralized and decentralized versions of $\mathcal{MNA}$. Subsequently, Section 5.3 illustrates the empirical evaluation of our proposed approach as opposed to the current state-of-the-art. Finally, we conclude the chapter with a summary in Section 5.4.

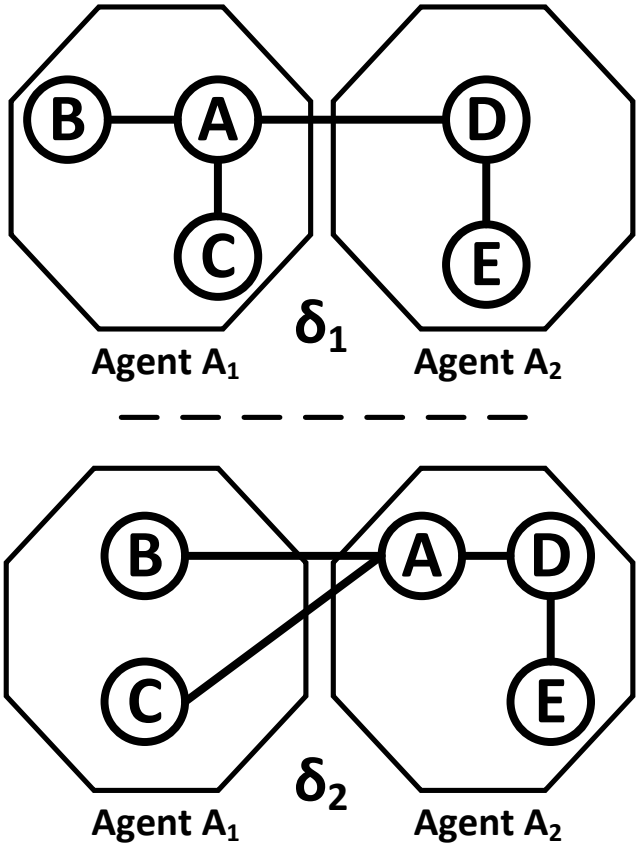
[1]See Section 2.5.4 for details of the applications' preference.

5.1 Problem Formulation

As mentioned in Section 2.1, a DCOP can be defined by a tuple $\langle X, D, F, A, \delta \rangle$, where X is a set of discrete variables $\{x_0, x_1, \ldots, x_m\}$ and $D = \{D_0, D_1, \ldots, D_m\}$ is a set of discrete and finite variable domains. Each variable x_i can take its value from the domain D_i. F is a set of constraints $\{F_1, F_2, \ldots, F_{\mathcal{L}}\}$, where each $F_i \in F$ is a function dependent on a subset of variables $\mathbf{x}_i \in X$ defining the relationship among the variables in $\mathbf{x}_i$. Thus, the function $F_i(\mathbf{x}_i)$ denotes the value for each possible assignment of the variables in $\mathbf{x}_i$. The dependencies between the variables and the functions are often graphically represented by a constraint graph such as a junction tree, factor graph or DFS tree, where the nodes (i.e. variables and/or functions) of the corresponding graphical representation G are being held by a set of agents $A = \{A_1, A_2, \ldots A_k\}$. This mapping of nodes to agents is represented by $\delta : \eta \rightarrow A$. Here, η stands for the set of nodes within the constraint graph G. As a result of the mapping represented by δ, we get a partition $P(A)$ of $k = |A|$ sub-graphs (i.e. $G_1, G_2, \ldots, G_k$) from G, where each $G_j \in G$ is held by the agent $A_j \in A$ (Equation 5.1).

$$P(A) \leftarrow \bigcup_{j=1}^{k} G_j \mid \forall j' \neq j : (G_j \cap G_{j'}) = \emptyset \tag{5.1}$$

Within this model, a GDL-based DCOP algorithm operates directly on G by passing messages among the nodes $\eta \in G$ to have each agent assign values to its associated variables from their corresponding domains. The aim is to maximize (minimize) the aggregated global objective function which eventually produces the value of each variable, $X^* = \arg\max_X \sum_{i=1}^{\mathcal{L}} F_i(\mathbf{x}_i)$. In such algorithms, to compute a message for a particular neighbour, a node takes into account the messages from its neighbours along with its own utility. Thus, a number of nodes initially start generating (i.e. computation) and then sending (i.e. communication) messages, each of which we jointly denote as a single event. That means, an event involves both the computation and the communication of a certain message. In this process, the completion of certain events might trigger one or more new events to be initiated. Thus, the total message passing procedure will complete when each node receives messages from all of its neighbours, such that all the running events are completed without initiating any new events. The dependencies among the events during the message passing process can be seen as an event-based dependency graph $E_G(\mathbb{A}, P)$, where $\mathbb{A}$ is the specific GDL-based DCOP algorithm deployed and P is the partition obtained from Equation 5.1. Formally, let E be the set of events $\{E_1, E_2, \ldots, E_l\}$ of $E_G(\mathbb{A}, P)$. Here, the weight of an edge $E_i \rightarrow E_j$ between two events E_i and E_j represents the time required to complete event E_i. Finally, the longest path cost of all existing event pairs is the total completion time $T(\mathbb{A}, P)$ for a given graphical representation of a DCOP (Equation 5.2). Here, the function $\upsilon(E_i, E_j)$ represents the time elapsed (i.e. path cost) between the starting of the event E_i and the end of the event E_j.



Computation Cost	
Node	**Time-unit**
A	25
B	0
C	0
D	12
E	0

Communication Cost	
Type	**Time-unit**
Inter-agent	7
Intra-agent	2

Figure 5.1: Two sample mappings of nodes $\{A, B, C, D, E\}$ of a constraint graph to agents A_1 and A_2. In the figure, nodes are denoted by circles and agents as octagons.

$$T(\mathbb{A}, P) = \max_{\forall_{E_i, E_j} \in E_G(\mathbb{A}, P)} \upsilon(E_i, E_j) \tag{5.2}$$

In this formulation, without loss of generality, we assume each agent possesses its own memory and a separate processing unit[2]. Here, on behalf of the sending node of an event, the holding agent generates and then sends the message to the receiving node. The sending node and its corresponding receiving node can either be held by the same agent or by two different agents. The time required to send a message in the former case can be termed the intra-agent communication cost and the latter the inter-agent communication cost. The former is typically less expensive in terms of communication cost than the latter (Sultanik et al., 2008). This is because it requires less time for an agent to take a message from its local memory than from a memory belonging to a different agent. Moreover, since an agent has a single processing unit, it cannot compute more than one message at a time. However, it can compute a message while transmitting another one and vice versa. As a consequence, allowing an agent to hold too many nodes

[2]In a multi-processing capable setting, each processing unit with separate memory can be considered as an agent.

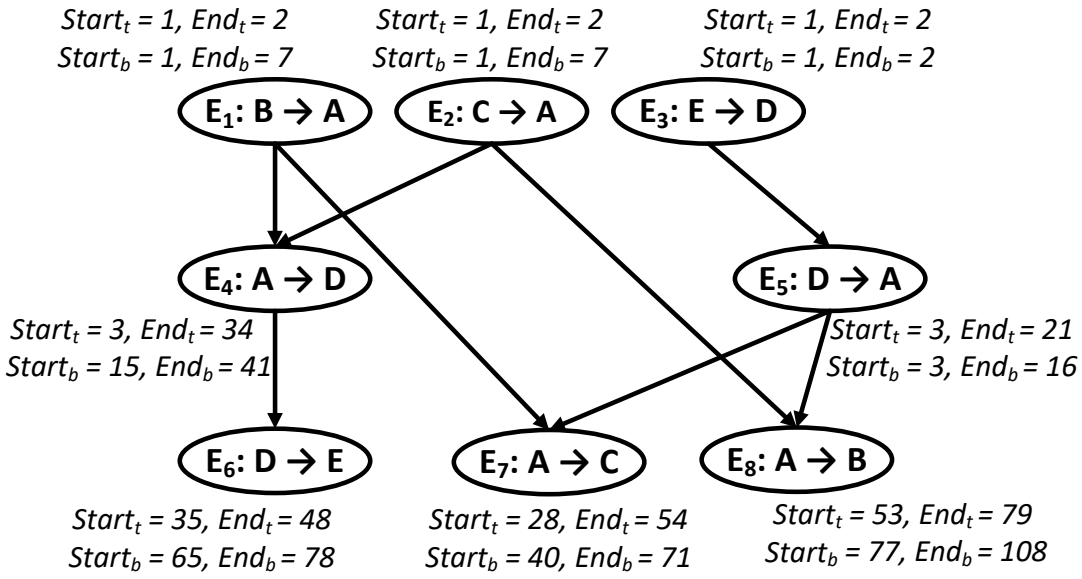


Figure 5.2: Event-based dependency graph for the constraint graph of Figure 5.1.

eventually increases the waiting time for the nodes within the agent. Considering this trade-off, the ultimate objective is to minimize the completion time $T(\mathbb{A}, P)$ of a message passing algorithm $\mathbb{A}$ by providing an efficient mapping of nodes to agents (Equation 5.3).

$$P^* = \arg\min_{P} T(\mathbb{A}, P) \tag{5.3}$$

Figure 5.1 illustrates two sample assignments of a constraint graph having five nodes $\{A, B, C, D, E\}$ between two agents A_1 and A_2. On the one hand, two sets of nodes $\{A, B, C\}$ and $\{D, E\}$ are being held by the agents A_1 and A_2 respectively in the mapping δ_1, depicted at the top of Figure 5.1. On the other hand, A_1 holds nodes $\{B, C\}$ and A_2 holds nodes $\{A, D, E\}$ in the mapping δ_2, shown at the bottom of that figure. Additionally, the message computation cost of each node and the message transmission/communication cost for the edges in terms of time-units are given in the tables on the right side of Figure 5.1. As can be seen, the computation cost of node A is 25 time-units, meaning node A requires 25 time-units to generate a message for any one of its neighbours. In this example, the inter-agent and the intra-agent communication cost is 7 and 2 time-units, respectively. Thus, the sending node A requires 7 time-units to send a message to the receiving node D when both A and D are being held by different agents (δ_1). Otherwise, the same message takes 2 time-units, as is the case in mapping δ_2.

The reason why the efficient mapping of node-to-agent is significant can be clearly seen from Figure 5.2, where we generate an event-based dependency graph of the message passing for the exemplar constraint graph shown in Figure 5.1. Here, the starting and

finishing time of each event are represented by $Start_t/Start_b$ and End_t/End_b respectively, where t stands for the mapping δ_1 and b corresponds to δ_2. Finally, the largest value of End_t and End_b represents the completion time of the constraint graph based on the mappings δ_1 and δ_2, respectively. In this particular example, we get 8 events: $\{E_1, E_2, \ldots, E_8\}$. For instance, event E_4 stands for the summation of the computation and the communication time of the message sending from A to D and the event E_4 can only initiate after events E_1 and E_2 have finished, that is when node A receives messages from nodes B and C. It is worth mentioning that if the holding agent of node A (i.e. the sending node of event E_4) is already computing a message for another node, then E_4 has to wait until the agent finishes computing the message, even if E_1 and E_2 have finished. Significantly, nodes A and D have degrees higher than that of any other nodes in the constraint graph, and as such, they require substantially more time-units to generate each of their messages. In the mapping δ_2, both A and D are being held by agent A_2. This potentially leads to a situation where the nodes of A_2 have to wait for a long period of time, even if the events they depend on have finished. In this worked example, events E_7 and E_8 have to wait for an additional 24 and 61 time-units respectively, even though they are ready to compute (δ_2). On the other hand, the waiting times are 7 and 32 time-units respectively in δ_1 due to the fact that the higher-degree nodes A and D are held by two different agents. As a result, we observe that the completion time of a DCOP algorithm for the mapping δ_1 is 79 time-units, and 108 time-units for δ_2. Thus, even for a small constraint graph of 5 nodes, it is possible to save around 27% of completion time through an efficient mapping of node-to-agent.

However, finding an optimal mapping is an NP-hard problem (see Section 2.5.4). Consider an example where a constraint graph of 25 nodes have to distribute among 8 agents. In this case, there are $1,081,575$ possible uniform mappings. In addition to that, we cannot ignore the possibility of getting better results from a non-uniform assignment. Even though the search space can be reduced by giving preference to the contiguous nodes being held by the same agent, the number is still significant (see empirical results in Section 5.3). Furthermore, the optimal mapping is completely dependent on the structure of the graph, so it is not possible to predict such mappings in advance based on prior information. Under such circumstances, finding an optimal mapping is not practicable for large multi-agent settings. This leads us to the $\mathcal{MNA}$ heuristic detailed in the following section.

5.2 The $\mathcal{MNA}$ Heuristic

Considering the observations made in the previous section, $\mathcal{MNA}$ specifically aims to find mappings where nodes with high degrees are held by different agents. In other words, the objective is to obtain a node-to-agent mapping for a DCOP, where nodes with higher computational requirements for producing their messages do not end up being held by the same agent. At the same time, it is important to ensure that the

mapping process itself is not prohibitively expensive in terms of time consumption. To this end, we propose two versions of $\mathcal{MNA}$, centralized and decentralized, each of which is discussed in Sections 5.2.1 and 5.2.2, respectively.

5.2.1 Centralized Version of $\mathcal{MNA}$

The complete process of $\mathcal{MNA}$'s centralized version is detailed in Algorithm 8. As aforementioned, it aims to reach a point where no two high-degree nodes are held by the same agent. Subsequently, a suitable agent is picked for each of the remaining nodes of a DCOP graphical representation based on this initial assignment. $\mathcal{MNA}$ operates directly on the corresponding graphical representation G of a DCOP that is going to be solved by deploying a GDL-based algorithm $\mathbb{A}$. Here, G consists of a set $\eta = \{\eta_1, \eta_2, \dots \eta_N\}$ of N nodes and a set $A = \{A_1, A_1, \dots A_k\}$ of k agents. At the end, Algorithm 8 returns $\delta : \eta \rightarrow A$, that is the mapping δ of the nodes η to their associated agents A. In line 1, a set $deg = \{deg(\eta_1), deg(\eta_2), \dots, deg(\eta_N)\}$ represents the number of connected neighbours of the nodes in η. More specifically, the function $deg(\eta_i) \in deg$ stands for the number of neighbours of the node $\eta_i \in \eta$, and it also provides information regarding how many incoming messages are required to produce each of η_i's outgoing messages, taking the deployed algorithm $\mathbb{A}$ into consideration. Then, line 2 presents the set of domains D, and each $D_i \in D$ is a finite set containing the values from which its associated node η_i has to take its preferred value. It is clearly illustrated in the example of the previous section that the degree of each node and the domain sizes of the connected neighbouring nodes contribute significantly in determining the overall completion time for a particular mapping. To be exact, the computation cost of the node η_i in terms of time corresponds to the values of $deg(\eta_i)$ and D_i. In the worked example of Figure 5.1, the degrees of node A and B are 3 and 1, respectively. Therefore, node A has to consider the messages of at least two nodes along with its own utility to generate a message for any of its neighbours. Moreover, the time required to generate a message is highest for node A, as its degree is higher than that of any other nodes. On the other hand, node B only needs to send a message to its only neighbouring node A. Consequently, for B to be able to generate that message, it does not need to rely on receiving any other message. As a result, B can immediately generate the message based on its local utility or often this is a pre-defined initial message. Thus the computation cost of B is negligible. Afterwards, line 3 computes the value of $uniformVal$, which is the ratio of the number of nodes N and the number of agents k in G.

It is noteworthy that the problem of node-to-agent mapping becomes trivial if all the nodes possess similar degrees and equal domain size. In this case, we can uniformly distribute the nodes among the agents by giving preference to the contiguous nodes being held by the same agent (lines $4-5$). Nevertheless, this is not the case for most DCOP applications, rather it is common to have nodes with dissimilar degrees and

Algorithm 8: $\mathcal{MNA}$ (G, η, A, $\mathbb{A}$)

Input: G is the corresponding graphical representation of a DCOP consisting of a set $\eta = \{\eta_1, \eta_2, \ldots \eta_N\}$ of N nodes and $A = \{A_1, A_1, \ldots A_k\}$ is the set of k agents participating in the optimization process, where $k <= N$. $\mathbb{A}$ stands for the deployed GDL-based DCOP algorithm.

Output: Mapping δ of the nodes of η to their associated agents A (i.e. $\delta : \eta \rightarrow A$), so that overall completion time can be minimized. Note that, each node can be held by a single agent; however, each agent can hold several nodes.

Let $deg = \{deg(\eta_1), deg(\eta_2), \ldots, deg(\eta_N)\}$ be the set where each $deg(\eta_i) \in deg$ stands for the degree/number of connected nodes of η_i

D is a set of domains $\{D_1, D_2, \ldots, D_N\}$, where each $D_i \in D$ is a finite set containing the values from which its associated node η_i has to get its preferred value

$uniformVal \leftarrow N/k$

if $(deg(\eta_i) - deg(\eta_{i'}) == 0) \wedge (|D_i| - |D_{i'}| == 0), where\, \forall \eta_i, \eta_{i'} \in \eta,\, \forall D_i, D_{i'} \in D$ **then**

```
// Contiguous uniform node-to-agent mapping, when the nodes possess
   similar degree and equal domain size.
```

  return $\delta_{uniformVal} : \eta \rightarrow A$

else

  $\lambda \leftarrow k\text{-}largestNodes(\mathrm{G}, \eta, k)$ `// Find a set` λ `of` k `largest nodes from` η `in terms of degree. Use the domain size of the connected nodes in case of a tie.`

  $\delta : \lambda \rightarrow A$ `// Distribute the nodes of` $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ `to` A `such that each agent holds a single node.`

  $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ are the control points of the graph G

  foreach *node* $\eta_i \in \eta \setminus \lambda$ **do** `// Distribute non control-point nodes`

    $\lambda_{cp} \leftarrow minDistance(\mathrm{G}, \eta_i, \lambda, uniformVal),\, where\, \lambda_{cp} \in \lambda$ `// Call Algorithm 9: choose the suitable control point` λ_{cp} `for the node` η_i.

    $\delta : \eta_i \rightarrow \lambda_{cp}.A_{cp}$ `// allocate` η_i `to the agent` A_{cp} `that holds the control point` λ_{cp}.

  return $\delta : \eta \rightarrow A$

Algorithm 9: minDistance($G, \eta_i, \lambda, uniformVal$)

Input: λ is a set of control points of the graph G, η_i is a non-control point node of G to be associated with one of the control point nodes of λ and $uniformVal$ is obtained from line 3 of Algorithm 8.

Output: $\lambda_m \in \lambda$, the corresponding control point for η_i.

```
1  λ' ← λ
2  λ_m ← sPath(G, η_i, λ')
3  if p(λ_m, A_m) < uniformVal then    // when the agent A_m corresponds to λ_m
                                       holds fewer nodes than the value of uniformVal.
4      return λ_m
5  else
6      λ' ← λ' \ λ_m
7      if λ' ≠ ∅ then
8          go to line 2
9      else
10         λ_m ← alt_sPath(G, η_i, λ)      // assign η_i to the closest control point
                                              that does not currently associates the most number of non-control
                                              point nodes among λ.
11         return λ_m
```

domain size (Kim & Lesser, 2013; Leite et al., 2014). This phenomenon, particularly, accounts for the differences in completion time for various possible mappings of nodes to agents. Specifically, lines $6-13$ of the algorithm concentrate on this issue. Now, the function $k\text{-}largestNodes(G, \eta, k)$ finds the k largest nodes from η in terms of degree. In case of a tie, it uses larger domain size, then records them to a set $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ (line 7). As a result, we get top k nodes with the highest degrees in G that require more time-units to compute each of their messages. At this point, line 8 allocates each node $\lambda_i \in \lambda$ to the different agents of A, and $\mathcal{MNA}$ defines each of these nodes as a control point (explained shortly) of the constraint graph G (line 9). In other words, the set $\{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ of k high-degree nodes are going to act as the control points, each of which is exclusively held by one of the k agents of A. In the example of Figure 5.1, the agents A_1 and A_2 are participating in the optimization process, hence the value of k is two. Therefore, we need to find two control points from the set of nodes: $\{A, B, C, D, E\}$. In this particular instance, $\mathcal{MNA}$ picks A and D as the control points as they posses degrees that are higher than those of the other nodes, and they should be held by those two different agents. Let A and D be held by agents A_1 and A_2, respectively. This is significant because it assures that no two high-degree nodes will be held by the same agent, which is the biggest cause of an increase in the waiting time (as discussed in the previous section).

At this point, the for loop of lines $10-12$ associates the rest of the nodes that are not the control points (i.e. $\eta \setminus \lambda$), to their corresponding agents. In so doing, we utilize the concept of Fortune's algorithm to generate the Voronoi diagram (Fortune, 1987). Notably, a Voronoi diagram is a partitioning of a plane into regions based on the distance to a specific subset of points of the plane. This subset of points, denoted as control points, is specified beforehand. For each of the control points, Fortune's algorithm generates a corresponding region consisting of all points closer to the control point than to others. In other words, given a set of control points in a plane, Fortune's algorithm specifically finds the associated control points for the rest of the points on that plane, based on the nearest Euclidean distance at the worst case cost of only $O(N \log N)$ time. Here, the function $minDistance(\mathrm{G}, \eta_i, \lambda, uniformVal)$, detailed in the pseudo-code of Algorithm 9, takes as input a non-control point node η_i, the subset λ of η that acts as the control points and previously computed $uniformVal$, and then finds a suitable control point $\lambda_{cp} \in \lambda$ for η_i (line 11 of Algorithm 8).

The function is inspired by the method employed by Fortune's algorithm to obtain the appropriate control points for all such non-control point nodes. However, unlike Fortune's algorithm, which uses only the shortest Euclidean distance as the metric to choose the suitable control point for a node, $\mathcal{MNA}$ uses different criteria. This is because we have to deal with a graphical representation instead of a plane. In more detail, in line 2 of Algorithm 9, $sPath(G, \eta_i, \lambda')$ finds such a control point $\lambda_m \in \lambda'$ for η_i that possesses the shortest path from η_i within the constraint graph G, and G is considered as an unweighted graph during this process. Here, λ' is a stand-in for the set of control points λ (line 1). At this point, if the holding agent of λ_m, denoted by A_m, currently holds fewer nodes than the value of $uniformVal$, then λ_m becomes the desired control point for η_i (lines $3-4$). Here, the function $p(\lambda_m, A_m)$ represents the current number of nodes held by the agent A_m. If this is not the case, λ_m is excluded from λ', and the process is repeated (lines $6-8$). Now, if none of the control points of λ' satisfies the condition of line 3, we assign η_i to its closest control point that does not already associate the most number of non-control point nodes among all the control points λ (lines $9-11$). This is important because in this way we can ensure that no agent corresponding to a control point ends up holding too many nodes. Notably, in case of a tie in either or both of the functions in lines 2 and 10, priority should be given to the control point whose associated agent possesses higher computational power. Thus, we can utilize the disparity in agents' computational capabilities (i.e. processing power). Hence, Algorithm 9 returns the control point to line 11 of Algorithm 8, which is denoted by λ_{cp}. Afterwards, line 12 assigns node η_i to the agent holding its associated control point λ_{cp}. As a result, we produce a mapping where a high-degree node is held by the same agent as its connected neighbours in most cases. Such a mapping experiences an additional axiomatic benefit; that is, the intra-agent messages greatly outnumber more expensive inter-agent messages. This is because the majority of the messages generated by the high-degree nodes are transmitted by means of the intra-agent communication.

In the example of Figure 5.1, the unweighted path cost (i.e. distance) of the non-control points nodes B, C and E from control point A are one, one and two, respectively. In contrast, the path costs are two, two and one respectively from control point D. According to the regulation of $\mathcal{MNA}$, nodes B and C will be associated with control point A, as they have the shortest path from A as opposed to D. Thus, along with node A, both nodes B and C are eventually held by agent A_1. In the same way, node E picks control point D, and both of them are held by agent A_2. Finally, the mapping obtained by following the process of $\mathcal{MNA}$ is δ_1 which significantly outperforms δ_2, as already illustrated in the explanation of Figure 5.2 (see Section 5.1). The time complexity of the $\mathcal{MNA}$ algorithm involves two parts. Firstly, $\mathcal{O}(k + (N - k) \log k)$ for finding the k-largest nodes (i.e. control points) from N nodes. Secondly, $\mathcal{O}(N \log N)$ for choosing suitable control points for the rest of the nodes in G. The overall complexity is therefore $\mathcal{O}(N \log N)$ as the value of k is always smaller (or in the worst case, equal) to the number of nodes N.

5.2.2 Decentralized Version of $\mathcal{MNA}$

Until this point, $\mathcal{MNA}$ considers those DCOP settings where a node-to-agent mapping is not included as a part of the problem definition, or considerable flexibility exists in choosing the mapping in a centralized manner. However, as discussed in Section 2.5.4, the assignment is assumed as a part of the problem in a number of applications, and as such, the centralized approach is not suitable for them. Moreover, it is important for $\mathcal{MNA}$ to cope with settings that are not impervious to the introduction of new nodes (and the departure of existing nodes), even after the node-to-agent mapping is done or given. In order to yield the benefits similar to that of the centralized version in such cases, we introduce a decentralized version of $\mathcal{MNA}$ (i.e. *Steps* $1 - 4$*)*. To be precise, this particular version of $\mathcal{MNA}$ can be used before initiating the message passing in applications where the mapping is given a priori; at the same time, it can be used in the event of a change within the graphical representation G during the runtime of a GDL-based algorithm.

- ***Step*** **1: Token Generation.** Each agent $A_j \in A$ generates a token that contains degree $deg(\eta_i)$ and domain info D_i for each node η_i it currently holds. The token also contains $cap(A_j)$, which represents the computation capability (i.e. processing power) of agent A_j.
- ***Step*** **2: Multicast Token.** Each agent A_j (or the agents that experience change in G at runtime) shares its token to agents holding nodes within the path distance of length l in G. To be able to ensure that contiguous nodes are being held by the same agent in most cases, it is recommended that the value of l is not too large.[3] Moreover, larger values of l would mean more messages are exchanged, thus eventually increasing overall communication costs.

[3]By considering the value of l within the range 3 to 5, we empirically observe a similar performance between decentralized $\mathcal{MNA}$ and it's centralized version.

- ***Step* 3: Request Message.** Based on the information of degree and domain from the received tokens, each agent A_j (or only the receiving agents in the event of change) decides whether it needs to hand over one or more nodes it is holding to some other agent(s). The decision should be taken based on the main feature of $\mathcal{MNA}$; that is, an agent should hold the least number of high-degree nodes. Note that, in the case of a tie, priority should be given to an agent that possesses higher processing power (i.e. $cap(A_j)$). Then, each of the deciding agents sends a single unicast request message to each of the agents it wants to relinquish its one or more nodes to.
- ***Step* 4: Response Message.** Finally, considering all the received *Request Messages*, an agent takes a decision (based on the main feature of $\mathcal{MNA}$ and $cap()$) about each node it received request(s) for. Then, it sends a message in response to each of the *Request Messages*, where the value 1 is used to mark the nodes it is willing to hold, and 0 is used otherwise.

In terms of complexity, concurrently, each agent A_j is observed to generate its own token, which is a small message that contains its nodes' degree, domain information and $cap(A_j)$ based on pre-existing data. Additionally, two decision operations are performed in *Step* $3-4$ of decentralized $\mathcal{MNA}$. Thus, the overall computation complexity is $\mathcal{O}(2)$, and in effect, negligible with regard to time. Nevertheless, in *Step* 2, the agent transmits the token (i.e. a small size message) to the holding agents of nodes within the path distance l in G. Since the value of l and the token size is usually small, the overall communication complexity of this approach is linear in terms of time (see Figure 5.5 and its discussion for empirical evidence).

5.3 Empirical Evaluation

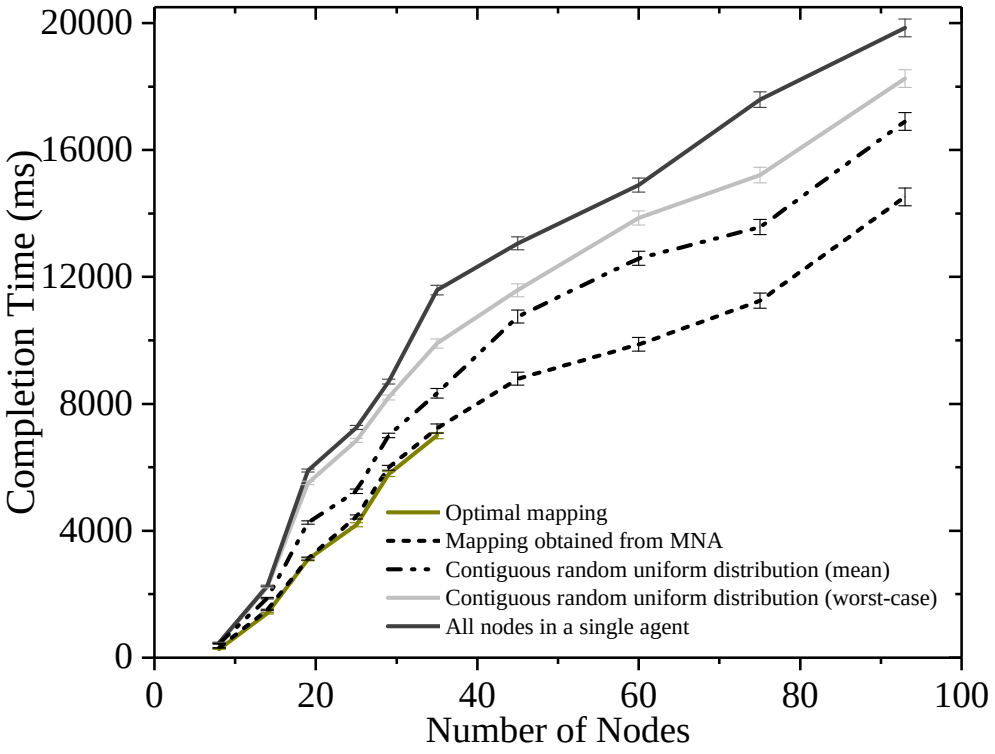
We now empirically evaluate the performance of $\mathcal{MNA}$[4] in terms of completion time, and compare it with the optimal mapping. As finding an optimal mapping is not feasible for large-scale settings (see Figure 5.5), we also compare $\mathcal{MNA}$ with two more benchmarks: (i) a centralized approach, where all the nodes are assumed to be held by a single agent, and (ii) a contiguous random uniform distribution (i.e. mapping). We choose the former as a benchmark to check whether distributing to many agents is indeed necessary. On the other hand, the latter checks the impact of doing this mapping in a simple way, similar to the method used by SECP (see Section 2.5.4). All the experiments were performed on a simulator in which we generated different instances of the constraint graph that have a varying number of nodes from 7 to around 100, and the degree of each node is randomly chosen from the range 1 to 7. In the simulation, we made use of the so-called "event-based dependency graph" method (see Section 5.1 for details) to obtain the completion time for a particular node-to-agent mapping of a constraint graph. In order to accomplish this, we performed an independent set of experiments to generate each node's computation cost (i.e. time) in advance. Here, we consider the domain size

[4]Note that both centralized and decentralized $\mathcal{MNA}$ provide comparable node-to-agent mapping, depending on the choice of the value l in the decentralized version (see Figure 5.4 for details).

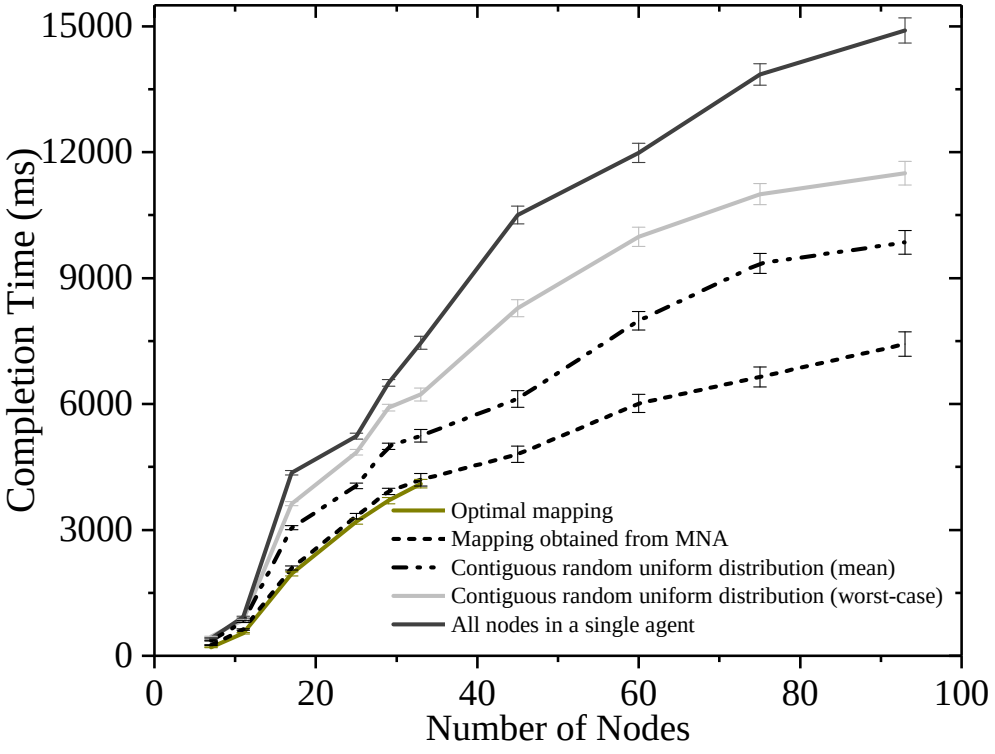
of all the nodes in the range of $11 - 30$. To obtain a node's computation time for all its messages, we initially generated 20 messages of varying sizes from that range, and then averaged the time elapsed in computing the messages. We did so to reflect the growth of search space in the generation of a message by an individual node with an increase in the node's degree and domain size in a DCOP (Farinelli et al., 2008; Kim & Lesser, 2013; Petcu, 2005). For example, the computation cost (i.e. the time required to generate each message) of a node with degree 5 is calculated by taking the average duration to compute 20 messages of following complexities: $(11^5, 12^5, \ldots, 30^5)$, where degree $n = 5$ and domain size $d = (11, 12, \ldots, 30)$. While these experiments were performed in a simulator, it is worth mentioning that we use the FRODO repository (Léauté et al., 2009) to generate utility (i.e. cost) tables of such complexities. Meanwhile, we used a network simulator tool (GNS3) in order to obtain the intra-agent and inter-agent communication costs in terms of time (Welsh, 2013). It has been observed that the former type of communication is a few times faster than the latter because we can take the underline network cost into account by using GNS3. Notably, we obtained the values (costs) of the parameters (i.e. computation and communication) through independent empirical observations (and in advance), so as to accurately report the comparative performance from different conceivable mapping approaches without being affected by any application-specific factors (e.g. hazardous communication in disaster response scenario). Moreover, the exact value of communication cost (in terms of time), which has a significant impact on the overall completion time of a DCOP algorithm, cannot be ascertained accurately in a simulated environment that runs on a single machine (or even a few machines), implying that this would not reflect the application-specific situation such as disaster response, sensor networks, etc. Therefore, we chose to carry out such controlled and systematic experiments wherein the results are neither affected nor generated by skipping several implementation and application-specific issues. Without loss of generality, the comparative results are reported for a single round of message passing[5] for the constraint graphs with cycles, since the results for a single round stand-in as a proportional representation for multiple rounds required for the cyclic constraint graphs in this experiment. Nevertheless, we consider the total completion time of the message passing operation for acyclic graphs, as they converge after a single round of message passing. Finally, we report the results averaged over 20 test runs in Figure 5.3, recording standard errors to ensure statistical significance. Note that the simulator is being implemented and run in an Intel $i7$ Quadcore 3.4GHz machine with 16GB of RAM.

Figure 5.3(a) illustrates the comparative results considering the number of nodes-agents ratio from the range $(2-12)$. We found that the results are comparable for settings with higher node-agent ratios. The completion time considering the obtained mapping from $\mathcal{MNA}$ is compared with the optimal mapping for a particular constraint graph. To report the optimal result for the constraint graph, we run our simulation for all possible uniform mappings. Note that the results depicted in Figure 5.3 do not include the time

[5]We report results based on the standard message passing protocols used by GDL-based DCOP algorithms for our experiments. These results are comparable for message passing protocol used by DPOP and Action-GDL.



(a) Random constraint graphs



(b) Scale-free constraint graphs.

Figure 5.3: Empirical results for different instances of the constraint graphs with the number of nodes and the number of agents ratio: $(2-12)$. Error bars are calculated using standard error of the mean.

required to run the mapping algorithm ($\mathcal{MNA}$ or optimal) itself, but rather illustrate the completion time of the message passing based on the obtained mapping. We discuss the run-time of the algorithms shortly. In Figure 5.3(a), the dark yellow line indicates the time to complete the message passing from the optimal mappings. As finding such optimal results through this exhaustive approach is not practicable for larger settings, we can only report this up to the constraint graph of 35 nodes (see Figure 5.5). Here, the dashed black line represents the completion time on the mapping obtained from $\mathcal{MNA}$. Significantly, $\mathcal{MNA}$ always performs at a level of at least 90% of the optimal one. Moreover, in a number of instances, we observe that $\mathcal{MNA}$ provides the optimal performance. The solid black line represents the outcome from the centralized system, where a single agent holds all the nodes. It performs worse in all the instances because of the fact that an agent cannot compute more than a single message at a time. Even though all the communications are intra-agent in this case, the waiting time for the nodes eventually increases with the growth of the number of nodes. Afterwards, the dashed-dot-dot black line of Figure 5.3(a) shows the results of the mean of $10 - 50$ randomly taken contiguous uniform mappings for each constraint graph. As observed, $\mathcal{MNA}$ takes around $17 - 32\%$ less time compared to this benchmark for the constraint graphs having a number of nodes ranging from 7 to 35. Furthermore, we report 16% to around 23% performance gain of $\mathcal{MNA}$ in the larger constraint graphs compared to the same benchmark. Finally, the solid grey line reports the worst case outcome from the randomly taken contiguous uniform distributions to indicate the possible impact of doing this mapping in a trivial way. In the worst case, a randomly taken uniform distribution performs 25% to around 38% (i.e. around 1.23 to 1.6 times) slower than the mapping obtained from $\mathcal{MNA}$.

The same experiments were performed with scale-free graphs (Barabási et al., 1999), and Figure 5.3(b) illustrates those results. Although the results are comparable for both types of constraint graphs, we found a notable difference for larger settings. The performances of $\mathcal{MNA}$ compared to the contiguous random uniform distributions are better (i.e. 24% to around 33% for contiguous random uniform distributions (mean), and 30% to around 43% for the worst case) than what we observed in Figure 5.3(a) for the constraint graphs of around 70 nodes or more. This is because the degree distribution of a scale-free graph follows a power law that allows a small subset of nodes to possess much higher degrees than the rest of the nodes in a graph. This phenomenon is particularly suitable for $\mathcal{MNA}$ to obtain a good node-to-agent mapping. In addition, for both of these experiments, we run the one-way ANOVA with post-hoc Tukey HSD test. While doing so, we consider 4 heuristics (i.e. $\mathcal{MNA}$, contiguous random uniform distribution (mean), contiguous random uniform distribution (worst-case) and completely centralized approach) as treatments, each of which illustrates its performance compared to the optimal mapping. For each experiment, the observed p-value corresponding to the F-statistic of one-way ANOVA is lower than 0.05, suggesting that the one or more treatments are significantly different. Subsequently, we employ a post-hoc test (Tukey HSD) that also suggests that the performance of $\mathcal{MNA}$ is significantly different from each of the remains, individually (i.e. $p < 0.01$).

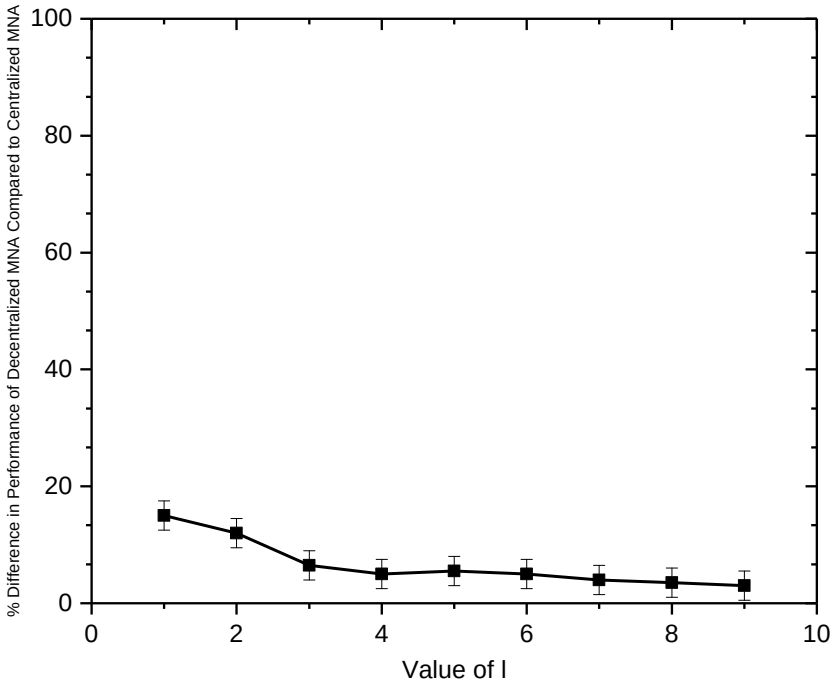
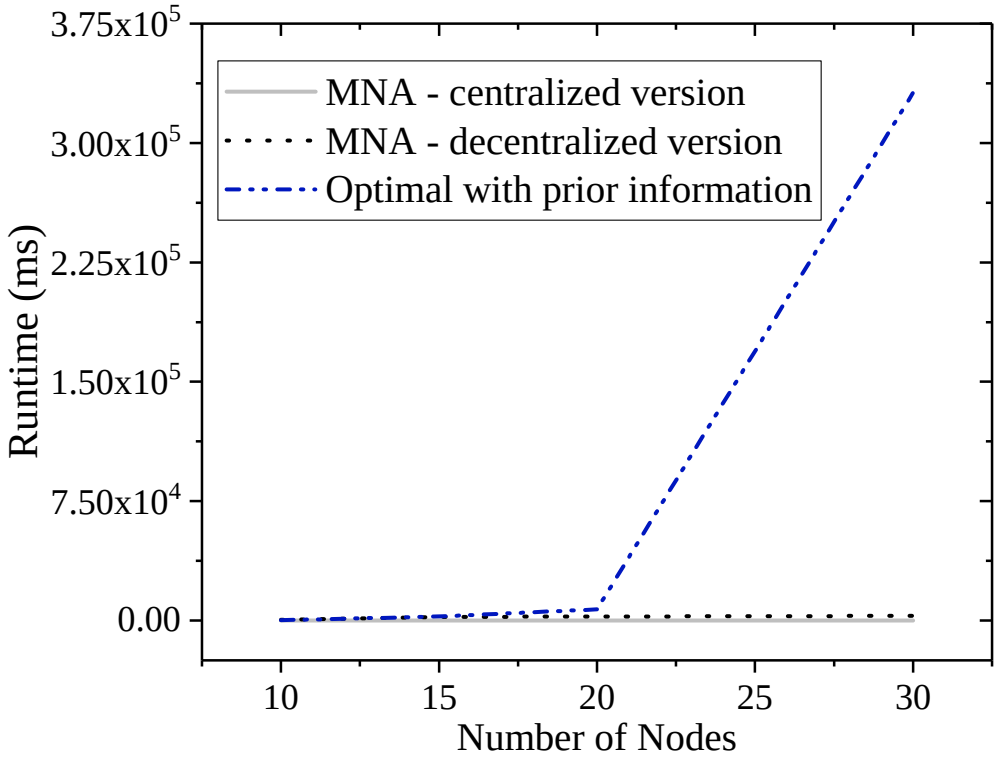
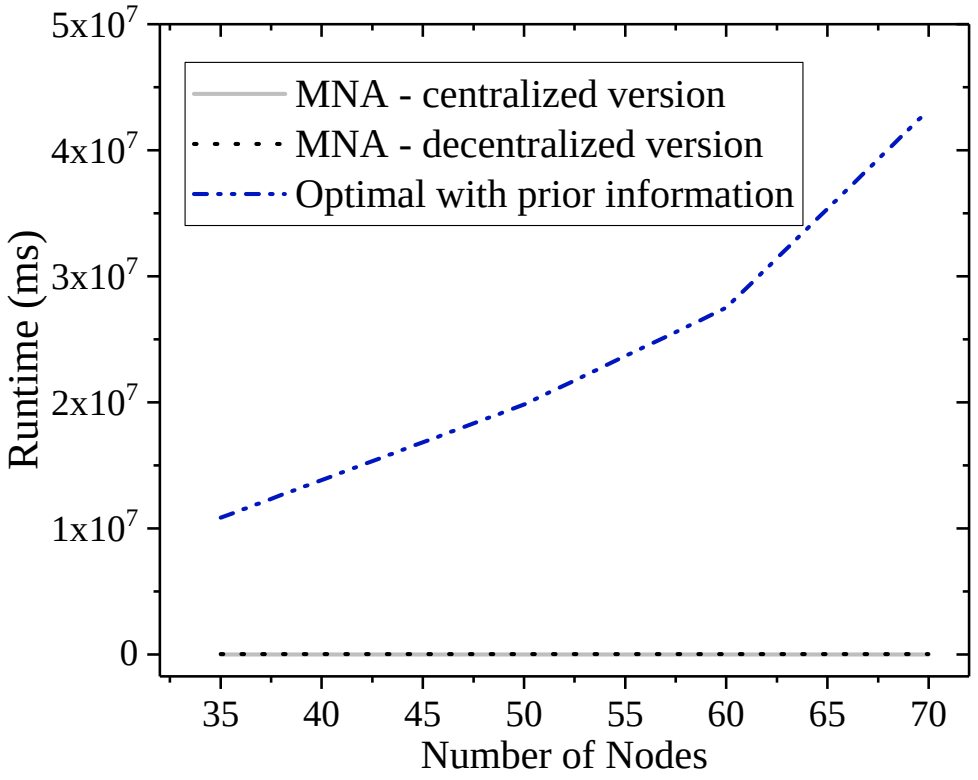


Figure 5.4: Differences in Decentralized $\mathcal{MNA}$'s performance as opposed to centralized $\mathcal{MNA}$ for different values of l (i.e. path distance). The reported results are calculated by taking average of 20 randomly generated constraint graphs based on the same setting as the centralized version. Error bars are calculated using standard error of the mean.

The aforementioned results clearly show a significant speed-up of message passing algorithms, when they are applied based on the mapping obtained from $\mathcal{MNA}$. Although we observe slight differences in the performance obtained using the decentralized version of $\mathcal{MNA}$ compared to the centralized one, it is apparent from Figure 5.4 that the difference is negligible, even for a value of l as small as 3. Nevertheless, we need to ensure that running $\mathcal{MNA}$ itself is not prohibitively expensive (since it is an additional pre-processing cost on top of the DCOP algorithms). To this end, we need to consider the time $\mathcal{MNA}$ (centralized and decentralized versions) actually takes to obtain the desired node-to-agent mapping for different constraint graphs, and compare this with the time to obtain the optimal mapping. To report the result for the decentralized version, we consider a processing unit with separate memory of a High Performance Computing (HPC) cluster as an agent. Specifically, Figures 5.5(a) and 5.5(b) show the results for the constraint graphs with the number of nodes ranging from 10 to 30 and 35 to 70, respectively. It is clear from the grey lines that the centralized version of $\mathcal{MNA}$ takes linear time to find the mapping for each of the constraint graphs. Although we observe that the decentralized version (dotted black lines) takes slightly more time than its centralized counterpart, it does not incur such delays that would make it prohibitively expensive to deploy. On the other hand, the results illustrated in dashed-dot-dot blue lines show that obtaining the optimal mapping is not practicable or is prohibitively expensive for



(a) Constraint graphs consist of $10 - 30$ nodes.



(b) Constraint graphs consist of $35 - 70$ nodes.

Figure 5.5: Comparative runtime to obtain the node-to-agent mapping: $\mathcal{MNA}$ vs Optimal.

the constraint graphs of around 25 nodes or more. Note that, to obtain the optimal mapping by considering all contiguous uniform distributions, we had to assume that the computation and communication cost of each messages are known in advance. This is generally unknown prior to executing an optimization algorithm. Taken together, finding an optimal mapping is practically infeasible, while the overall cost of $\mathcal{MNA}$ is linear.

5.4 Summary

Motivated by the fact that a good node-to-agent mapping entails a significant potential of reducing the overall completion time of a DCOP algorithm, we introduced centralized as well as decentralized versions of a node-to-agent mapping heuristic in this chapter. These versions can be implemented in both GDL-based exact and non-exact approaches. To this end, we formulate this specific phase of node-to-agent mapping as an optimization problem in such a manner that $\mathcal{MNA}$ can be applied to all GDL-based algorithms operating on different graphical representations (e.g. junction tree, factor graph or DFS-tree). In the formulation, our objective is to find a feasible mapping that minimizes the completion time of the DCOP algorithm (e.g. DPOP, Action-GDL or Max-Sum) operating on this mapping. Finally, we empirically evaluate the performance of our approach in terms of completion time, showing that it does perform at a level of around $90\% - 100\%$ of the optimal mapping, which is computationally infeasible to obtain in practice. Our results also denote an acceleration of $16\% - 40\%$ as compared to the state-of-the-art, implying that a message passing algorithm can perform $1.2 - 1.7$ times faster when using $\mathcal{MNA}$-generated node-to-agent mappings. Finally, we empirically illustrate that the speed-up can be attained with the expense of a linear run-time cost, which is significant given that an optimal mapping is indeed prohibitively expensive. These results are referring to the fact that the research challenge **C3** is addressed successfully.

Chapter 6

Conclusions and Future Work

This thesis develops a number of new approaches to accelerate three potentially expensive phases of GDL-based DCOP algorithms in order to improve their scalability and practical applicability. In Section 6.1, we present an overview of the contributions of this thesis within the overarching theme of the research requirements and challenges outlined at the beginning of the thesis. Finally, we give directions for future work in Section 6.2.

6.1 Conclusions

In this thesis, we sought to improve the scalability of existing DCOP algorithms without affecting their solution quality. This overall objective was influenced by the hypothesis that it is possible to improve the scalability of an existing DCOP algorithm by minimizing the total completion time of the algorithm. In line with this hypothesis, we explicitly identified computation and communication costs as two key elements of the algorithms that have a significant influence on increasing their completion time. We also identified that, while the reduced completion time of a DCOP algorithm is necessary to effectively implement it on a large and complex multi-agent system, preserving its solution quality poses a very stiff challenge. Thus, it is paramount to maintain solution quality and general applicability when speeding up these algorithms. As such, at the beginning of the thesis, we outlined our research requirements after taking these issues into consideration. Thereafter, we explored the strengths and weaknesses of the different classes of DCOP algorithms in light of the research requirements, and determined that GDL-based non-exact algorithms are well suited to fulfilling them. In addition, we pointed out that scalability continues to pose a challenge for this class of DCOP algorithms due to a number of potentially expensive phases. The main intuition behind the contributions of this thesis is attributed to this observation. In light of this, and

when contemplating on the above mentioned hypothesis along with the issue of maintaining solution quality during the process, we outlined three main research challenges (as discussed in Section 1.1), each of which has been addressed in Chapters 3, 4 and 5, respectively. It is noteworthy that, although none of these three main contributions are dependent on each other, any combination of them can be applied in the same algorithm to speed up the phases they correspond to.

In Chapter 2, we analysed the academic literature related to our specific area of work. Notably, we did not focus on a specific application domain; instead, we focused on generic DCOP solutions that can be applied to any multi-agent setting. In this context, we formally delineated the generic DCOP framework before discussing various exact and non-exact approaches that have been proposed to solve DCOPs. In doing so, we initially described three main graphical representations (i.e. DFS-tree, junction tree and factor graph) over which the DCOP algorithms operate, as well as the reasons for and against each of them. Then, we extrapolated on how non-exact approaches are more suitable than exact approaches in the context of solving large and complex real world DCOPs, despite in the fact that they compromise some solution quality. We further examined how GDL-based message passing non-exact algorithms are feasible options as opposed to other non-exact algorithms, and how this class of algorithms impart certain axiomatic benefits by using a factor graph as the graphical representation. We also presented the relevance of Max-Sum and Bounded Max-Sum algorithms, which are two of the most popular GDL-based algorithms in solving such problems. Subsequently, we discussed the expensive phases that these algorithms entail and explored the literature to examine existing approaches that have been proposed to speed-up those phases against the research requirements and challenges outlined in Chapter 1. In light of this discussion, we highlighted three phases of GDL-based algorithms which, in our view, hold the potential for considerable improvements.

In Chapter 3, we focused on the most expensive phase of GDL-based non-exact algorithms in terms of computation cost; that is, the maximization operation. In particular, we intended to speed-up these algorithms by reducing the computation cost of this operator. Moreover, we tried not to affect the solution quality and generic applicability of this algorithm in the process, which we jointly outlined as our research challenge **C1**. To address the challenge, we introduced a domain pruning algorithm, namely $\mathcal{GDP}$, to reduce the computation cost of the maximization operator whilst maintaining the same result, and that can be used regardless of any application dependency. We also provided a theoretical proof to support the claim with regards to the quality of solution. Moreover, we empirically observed a significant reduction of the operator's overall computation cost of around $33\% - 81\%$ for a DCOP, and that has been achieved at the expense of a quasi-linear runtime cost of $\mathcal{GDP}$. More importantly, we observed from our empirical evidence that the performance gain of $\mathcal{GDP}$ gets better with an increase in the

parameters upon which the maximization operator acts. Nevertheless, it is worth mentioning that $\mathcal{GDP}$ is only tailored for the maximization operator of non-exact GDL-based algorithms, thus not applicable to other DCOP algorithms that calculate maximization (if they have any) in a different way.

Having dealt with the expensive maximization operator, Chapter 4 turned to the second research challenge, denoted as **C2**, wherein the collective goal (i.e. speeding up, solution quality and generic applicability) is identical to the previous one. However, in this chapter, we addressed this challenge by speeding up the message passing process of GDL-based algorithms. We introduced a generic message passing protocol, namely $\mathcal{PMP}$, which significantly reduces the completion time of GDL-based algorithms that use factor graphs as the graphical representation. It is worth noting that during the process, $\mathcal{PMP}$ does not change the algorithms' overall outcome. To be precise, by replacing the currently used standard message passing protocol with $\mathcal{PMP}$, we observed a significant reduction in completion time for such algorithms, ranging from $37\% - 91\%$ depending on the scenario. During the course of attaining this performance, we took advantage of partial centralization and combined clustering with domain reduction as well as straight-line linear regression to determine the appropriate number of clusters for a given scenario. Here, the clustering process enables $\mathcal{PMP}$ to parallelize the message passing process, while the regression method makes $\mathcal{PMP}$ a deterministic approach that is able to split the original factor graph into a reasonably appropriate number of clusters. Furthermore, we empirically observed that around 98% (or more) of $\mathcal{PMP}$'s best possible outcomes can be achieved when operated on the number of clusters predicted by the straight-line linear regression.

Finally, Chapter 5 explores an important gap in the literature, namely the problem of finding good mappings of nodes to agents in DCOPs. Since the choice of assignment can significantly impact the completion time of the algorithms, it is imperative to find good assignments. We specially took cognisance of this insight in our final research challenge **C3** along with the issue of preserving the solution quality and generic applicability. To address this problem, we proposed $\mathcal{MNA}$, a near-optimal heuristic that provides an effective node-to-agent mapping for a DCOP in order to minimise the completion time of the optimization process. Moreover, with a view to apply $\mathcal{MNA}$ to all GDL-based algorithms running on different graphical representations, we formulated this particular phase of node-to-agent mapping as an optimization problem. Furthermore, we have proposed two versions (i.e. centralized and decentralized) of $\mathcal{MNA}$ to allow it to be used based on the suitability of application at hand. Finally, our empirical evidence is indicative of the fact that $\mathcal{MNA}$ performs at a level of around $90\% - 100\%$ of the optimal mapping, which is computationally infeasible to obtain in practice. Our results also show a speed-up of $16\% - 40\%$ as compared to the state-of-the-art, which is

attained at the expense of linear run-time cost of its own. It is worth mentioning that although we managed to produce the same solution from the deployed GDL-based DCOP algorithm, none of the contributions discussed in this thesis are capable of bettering the quality of its solution.

When taken together, the contributions of this thesis could be used to make existing GDL-based algorithms more scalable in that either they take less time to complete the internal operation (of a given size) of DCOP, or are adept at tackling a larger DCOP within the same completion time as a smaller one. In effect, they offer an opportunity to effectively deal with large real-world coordination problems formulated as DCOPs in various multi-agent application areas.

6.2 Future Work

The findings we have presented in this thesis have thrown up few new questions that need further investigation. In the remainder of this section, we discuss future work to extend the scope and applicability of our research.

- It is apparent from the discussion and empirical results presented in Chapter 3 that the development of $\mathcal{GDP}$ offers a major breakthrough in reducing the computation cost of the maximization operator of non-exact GDL-based DCOP algorithms. However, the main intuition of $\mathcal{GDP}$ has come from the message computation means of this particular class of non-exact algorithms. Therefore, this algorithm is not applicable to exact GDL-based solution approaches such as DPOP, Action-GDL and their variants. There are two main reasons why this limitation should be dropped. Firstly, each of these exact algorithms also performs computationally expensive maximization operation(s) during its corresponding "*util propagation*" phase (see Section 2.3.3). Secondly, the messages produced during this phase may be exponentially large and have, therefore, also been expensive in terms of communication cost. Hence, with a view to overcome this limitation from $\mathcal{GDP}$, future work needs to be done to establish whether $\mathcal{GDP}$ can be tailored for those algorithms. Success in either doing so or introducing a new technique with a performance similar to $\mathcal{GDP}$ would mean a remarkable progress in the context of scaling up those exact algorithms. This is acknowledged as one of the long-standing challenges endured by DCOP algorithms that always produce optimal results.
- As discussed in Chapter 4, $\mathcal{PMP}$ is a partially centralized approach since a specific part (i.e. the intermediate step) of it needs to be done by the cluster heads. A general phenomenon for partially centralized approaches is that they trade privacy for higher scalability (Such et al., 2014). Thus, it is conceivable, and even likely, that $\mathcal{PMP}$ sacrifices some privacy. However, the term "privacy" is quite broad from the perspective of multi-agent systems. In this context, Léauté & Faltings

(2013) provided a good summary of the classification of privacy that distinguished between agent privacy, topology privacy, constraint privacy, and assignment/decision privacy. Considering these variants coupled with the fact that even completely decentralized approaches often experience some loss of privacy, the evaluation of privacy loss is a non-trivial task (Grinshpoun & Tassa, 2016; Léauté & Faltings, 2013). Recently, Tassa et al. (2015, 2017) proposed a privacy preserved algorithm for non-exact GDL-based approaches which preserves three types of privacy: topology privacy, constraint privacy, and assignment/decision privacy. However, they assume that all constraints are binary and each agent holds a single variable, which greatly limits several axiomatic benefits of this class of algorithms (see Section 2.4.2). Against this background, analysing $\mathcal{PMP}$ in the context of DCOP privacy would be an interesting line of future research.

- As discussed in Section 2.4.2, the GDL framework has gained enormous success in many non-MAS (i.e. single-agent) application areas such as iterative decoding and computer vision in the form of probabilistic belief propagation in the Max-Product algorithm. Moreover, even the Max-Sum algorithm has recently been employed as a biclustering approach, which is a renowned data mining technique used to profile gene expression in the domain of biological science (Denitto et al., 2017). In such applications, and many others besides, oftentimes a single high-performance machine is deployed to carry out all the computation of messages. A modern high-performance machine includes a Graphic Processing Unit (GPU), which is a multiprocessor device that offers hundreds of computing cores and a rich memory hierarchy for supporting graphical processing. During the last few years, a number of programming models (e.g. CUDA (Sanders & Kandrot, 2010) and OpenACC (Farber, 2016)) have been developed for enabling the use of the multiple cores of a GPU to accelerate non-graphical applications (the so-called General Purpose GPU (GP-GPU)). In addition, even in the MAS literature, we have found a few studies that use GP-GPUs to accelerate some parts of the DPOP algorithm (Fioretto et al., 2015, 2016). As such, it would be an interesting future work to determine whether $\mathcal{PMP}$ can be incorporated with the concept of GP-GPUs to speed-up the Max-Product and/or Max-Sum algorithms, since it parallelizes the message passing process. Following this, further comprehensive research should be undertaken to assess its performance in different application areas.

By meeting these challenges, the practical applicability and scalability of the approaches developed in this thesis can be further increased.

Bibliography

Aji, S. M. & McEliece, R. J. (2000). The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2), 325–343.

Ali, S., Koenig, S., & Tambe, M. (2005). Preprocessing techniques for accelerating the dcop algorithm adopt. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)* (pp. 1041–1048).: ACM.

Awerbuch, B. (1985). A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3), 147–150.

Barabási, A., Albert, R., & Jeong, H. (1999). Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1), 173–187.

Bowring, E., Pearce, J. P., Portway, C., Jain, M., & Tambe, M. (2008). On k-optimal distributed constraint optimization algorithms: New bounds and algorithms. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2 (pp. 607–614).: IFAAMAS.

Bowring, E., Tambe, M., & Yokoo, M. (2006). Multiply-constrained distributed constraint optimization. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)* (pp. 1413–1420).: ACM.

Burke, D. A. & Brown, K. N. (2006). Efficient handling of complex local problems in distributed constraint optimization. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*, volume 6 (pp. 701–702).: Citeseer.

Cerquides, J. B., Farinelli, A., Meseguer, P., & Ramchurn, S. D. (2013). A tutorial on optimization for multi-agent systems. *The Computer Journal*, 57(6), 799–824.

Chen, Z., He, C., He, Z., & Chen, M. (2017a). Bd-adopt: a hybrid dcop algorithm with best-first and depth-first search strategies. *Artificial Intelligence Review*, 48(4), 1–39.

Chen, Z., He, Z., & He, C. (2017b). An improved dpop algorithm based on breadth first search pseudo-tree for distributed constraint optimization. *Applied Intelligence*, 47(3), 1–17.

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann, first edition.

Denitto, M., Farinelli, A., Figueiredo, M. A., & Bicego, M. (2017). A biclustering approach based on factor graphs and the max-sum algorithm. *Pattern Recognition*, 62, 114–124.

Elidan, G., McGraw, I., & Koller, D. (2006). Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the 22nd Conference on Uncertainty in AI (UAI)* (pp. 200–208).: AUAI.

Farber, R. (2016). *Parallel Programming with OpenACC*. Morgan Kaufmann, first edition.

Farinelli, A., Rogers, A., Petcu, A., & Jennings, N. R. (2008). Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2 (pp. 639–646).: IFAAMAS.

Farinelli, A., Vinyals, M., Rogers, A., & Jennings, N. R. (2013). Distributed constraint handling and optimization. In G. Weiss (Ed.), *Multi-Agent Systems,* chapter 12, (pp. 547–584). MIT Press, second edition.

Fioretto, F., Le, T., Pontelli, E., Yeoh, W., & Son, T. C. (2015). Exploiting gpus in solving (distributed) constraint optimization problems with dynamic programming. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP)*: Springer.

Fioretto, F., Pontelli, E., & Yeoh, W. (2018). Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61, 623–698.

Fioretto, F., Yeoh, W., & Pontelli, E. (2016). A dynamic programming-based mcmc framework for solving dcops with gpus. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP)* (pp. 813–831).: Springer.

Fioretto, F., Yeoh, W., & Pontelli, E. (2017). A multiagent system approach to scheduling devices in smart homes. In *Proceedings of the 16th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)* (pp. 981–989).: IFAAMAS.

Fitzpatrick, S. & Meertens, L. (2003). Distributed coordination through anarchic optimization. In V. Lesser (Ed.), *Distributed Sensor Networks: A Multi-Agent Perspective,* chapter 11, (pp. 257–295). Springer, first edition.

Fortune, S. (1987). A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4), 153–174.

Gallager, R. G., Humblet, P. A., & Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems*, 5(1), 66–77.

Grinshpoun, T. & Tassa, T. (2016). P-syncbb: A privacy preserving branch and bound dcop algorithm. *Journal of Artificial Intelligence Research*, 57, 621–660.

Gutierrez, P. & Meseguer, P. (2010). Saving redundant messages in bnb-adopt. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence* (pp. 1259–1260).: AAAI Press.

Han, J., Pei, J., & Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier, third edition.

Hirayama, K. & Yokoo, M. (2005). The distributed breakout algorithms. *Artificial Intelligence*, 161(1-2), 89–115.

Jarvis, D., Jarvis, J., Rönnquist, R., & Jain, L. C. (2013). Multi-agent systems. In *Multi-Agent Systems and Applications,* chapter 1, (pp. 1–12). Springer.

Jennings, N. R. & Wooldridge, M. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 115–152.

Kiekintveld, C., Yin, Z., Kumar, A., & Tambe, M. (2010). Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 1 (pp. 133–140).: IFAAMAS.

Kim, Y. & Lesser, V. (2013). Improved max-sum algorithm for dcop with n-ary constraints. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* (pp. 191–198).: IFAAMAS.

Kschischang, F. R., Frey, B. J., & Loeliger, H. A. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2), 498–519.

Kutner, M. H., Nachtsheim, C., & Neter, J. (2004). *Applied linear regression models*. McGraw-Hill/Irwin, fourth edition.

Léauté, T. & Faltings, B. (2013). Protecting privacy through distributed computation in multi-agent decision making. *Journal of Artificial Intelligence Research*, 47, 649–695.

Léauté, T., Ottens, B., & Szymanek, R. (2009). FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)* (pp. 160–164). `https://frodo-ai.tech`.

Leite, A. R., Enembreck, F., & Barthès, J. A. (2014). Distributed constraint optimization problems: Review and perspectives. *Expert Systems with Applications*, 41(11), 5139–5157.

Lesser, V. & Corkill, D. (2014). Challenges for multi-agent coordination theory based on empirical observations. In *Proceedings of the 13th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)* (pp. 1157–1160).: IFAAMAS.

Macarthur, K. (2011). *Multi-agent Coordination for Dynamic Decentralised Task Allocation*. PhD thesis, University of Southampton.

Macarthur, K. S., Stranders, R., Ramchurn, S. D., & Jennings, N. R. (2011). A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence* (pp. 701–706).: AAAI Press.

Maheswaran, R. T., Pearce, J. P., & Tambe, M. (2004a). Distributed algorithms for dcop: A graphical-game-based approach. In *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS)* (pp. 432–439).

Maheswaran, R. T., Tambe, M., Bowring, E., Pearce, J. P., & Varakantham, P. (2004b). Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 1 (pp. 310–317).: IEEE Computer Society.

Mailler, R. & Lesser, V. (2004). Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 1 (pp. 438–445).: IEEE Computer Society.

Mailler, R. & Lesser, V. R. (2006). Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 25, 529–576.

Modi, P. J., Shen, W., Tambe, M., & Yokoo, M. (2005). Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1), 149–180.

Müller, J. P. & Fischer, K. (2014). Application impact of multi-agent systems and technologies: a survey. In O. Shehory & A. Sturm (Eds.), *Agent-Oriented Software Engineering,* chapter 3, (pp. 27–53). Springer, first edition.

Pearce, J. P. & Tambe, M. (2007). Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *Proceedings of the 20th international Joint Conference on Artificial Intelligence (IJCAI)* (pp. 1446–1451).: AAAI Press.

Pecora, F., Modi, P., & Scerri, P. (2006). Reasoning about and dynamically posting n-ary constraints in adopt. In *Proceedings of the AAMAS'06 Distributed Constraint Reasoning Workshop (DCR'06)* (pp. 1–15).

Peri, O. & Meisels, A. (2013). Synchronizing for performance-dcop algorithms. In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence (ICAART)*, volume 1 (pp. 5–14).: SCITEPRESS.

Petcu, A. (2007). *A class of algorithms for distributed constraint optimization.* PhD thesis, Ecole Polytechnique Federale de Lausanne.

Petcu, A. & Faltings, B. (2007). Mb-dpop: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 1452–1457).: AAAI Press.

Petcu, A., Faltings, B., & Mailler, R. (2007). Pc-dpop: A new partial centralization algorithm for distributed optimization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7 (pp. 167–172).: AAAI Press.

Petcu, A., F. B. (2005). A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 266–271).: AAAI Press.

Pujol-Gonzalez, M., Cerquides, J., Farinelli, A., Meseguer, P., & Rodriguez-Aguilar, J. A. (2015). Efficient inter-team task allocation in robocup rescue. In *Proceedings of the 14th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)* (pp. 413–421).: IFAAMAS.

Ramchurn, S. D., Farinelli, A., Macarthur, K. S., & Jennings, N. R. (2010). Decentralized Coordination in RoboCup Rescue. *The Computer Journal*, 53(9), 1447–1461.

Rogers, A., Farinelli, A., Stranders, R., & Jennings, N. (2011). Bounded approximate decentralised coordination via the max-sum algorithm. *Artificial Intelligence*, 175(2), 730–759.

Rollon, E. & Larrosa, J. (2014). Decomposing utility functions in bounded max-sum for distributed constraint optimization. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP)* (pp. 646–654).: Springer.

Rust, P., Picard, G., & Ramparany, F. (2016). Using message-passing dcop algorithms to solve energy-efficient smart environment configuration problems. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 468–474).: AAAI Press.

Sanders, J. & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley, first edition.

Stefanovitch, N., Farinelli, A., Rogers, A., & Jennings, N. R. (2011). Resource-aware junction trees for efficient multi-agent coordination. In *Proceedings of the 10th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 1 (pp. 363–370).: IFAAMAS.

Stranders, R., Farinelli, A., Rogers, A., & Jennings, N. R. (2009). Decentralised coordination of mobile sensors using the max-sum algorithm. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, volume 9 (pp. 299–304).: AAAI Press.

Such, J. M., Espinosa, A., & García-Fornes, A. (2014). A survey of privacy in multi-agent systems. *The Knowledge Engineering Review*, 29(3), 314–344.

Sultanik, E. A., Lass, R. N., & Regli, W. C. (2008). Dcopolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the 7th international Joint Conference on Autonomous Agents and Multi-Agent Systems: Demo Papers* (pp. 1667–1668).: IFAAMAS.

Tarlow, D., Givoni, I., & Zemel, R. (2010). Hop-map: Efficient message passing with high order potentials. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)* (pp. 812–819).: MLR Press.

Tassa, T., Grinshpoun, T., & Zivan, R. (2017). Privacy preserving implementation of the max-sum algorithm and its variants. *Journal of Artificial Intelligence Research*, 59, 311–349.

Tassa, T., Zivan, R., & Grinshpoun, T. (2015). Max-sum goes private. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 425–431).: AAAI Press.

Vinyals, M., Pujol, M., Rodriguez-Aguilar, J., & Cerquides, J. (2010). Divide-and-coordinate: Dcops by agreement. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 1 (pp. 149–156).: IFAAMAS.

Vinyals, M., Rodriguez-Aguilar, J. A., & Cerquides, J. (2009). Generalizing dpop: Action-gdl, a new complete algorithm for dcops. In *Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2 (pp. 1239–1240).: IFAAMAS.

Vinyals, M., Rodriguez-Aguilar, J. A., & Cerquides, J. (2011). A survey on sensor networks from a multiagent perspective. *The Computer Journal*, 54(3), 455–470.

Wainwright, M. J. & Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1–2), 1–305.

Welsh, C. (2013). *GNS3 network simulation guide*. Packt Publishers.

Yeoh, W., Felner, A., & Koenig, S. (2008). Bnb-adopt: An asynchronous branch-and-bound dcop algorithm. In *Proceedings of the 7th international Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2 (pp. 591–598).: IFAAMAS.

Yokoo, M. (2001). *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer Science & Business Media, first edition.

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

Yokoo, M. & Hirayama, K. (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS)* (pp. 401–408).: AAAI Press.

Zivan, R. & Peled, H. (2012). Max/min-sum distributed constraint optimization through value propagation on an alternating dag. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 1 (pp. 265–272).: IFAAMAS.

Zivan, R., Yedidsion, H., Okamoto, S., Glinton, R., & Sycara, K. (2014). Distributed constraint optimization for teams of mobile sensing agents. *Autonomous Agents and Multi-Agent Systems*, 29(3), 495–536.