N° d'ordre NNT : 2019LYSEM023

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'École des Mines de Saint-Étienne

École Doctorale N° 488
Sciences, Ingénierie, Santé

Spécialité de doctorat : informatique
Discipline : intelligence artificielle

Soutenue publiquement le 3 octobre 2019, par :
Pierre Rust

Autonomous and Spontaneous Coordination between Smart Connected Objects

Coordination spontanée et autonome entre objets intelligents connectés

---

Devant le jury composé de :

GLEIZES, Marie-Pierre	Professeur, Université de Toulouse	Rapporteure
MANDIAU, René	Professeur, Université Polytechnique Hauts-de-France, Valenciennes	Rapporteur
JAMONT, Jean-Paul	Professeur, Université Grenoble Alpes, Valence	Examinateur
RODRÍGUEZ-AGUILAR, Juan A.	Chercheur, IIIA-CSIC	Examinateur
PICARD, Gauthier	Professeur, École des Mines de Saint-Étienne	Directeur de thèse
RAMPARANY, Fano	Chercheur, Orange Labs	Co-encadrant de thèse

Spécialités doctorales	Responsables :	Spécialités doctorales	Responsables
SCIENCES ET GENIE DES MATERIAUX	K. Wolski Directeur de recherche	MATHEMATIQUES APPLIQUEES	O. Roustant, Maître-assistant
MECANIQUE ET INGENIERIE	S. Drapier, professeur	INFORMATIQUE	O. Boissier, Professeur
GENIE DES PROCEDES	F. Gruy, Maître de recherche	SCIENCES DES IMAGES ET DES FORMES	JC. Pinoli, Professeur
SCIENCES DE LA TERRE	B. Guy, Directeur de recherche	GENIE INDUSTRIEL	N. Absi, Maitre de recherche
SCIENCES ET GENIE DE L'ENVIRONNEMENT	D. Graillot, Directeur de recherche	MICROELECTRONIQUE	Ph. Lalevée, Professeur

EMSE : Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'État ou d'une HDR)

ABSI	Nabil	MR	Génie industriel	CMP
AUGUSTO	Vincent	CR	Image, Vision, Signal	CIS
AVRIL	Stéphane	PR2	Mécanique et ingénierie	CIS
BADEL	Pierre	MA(MDC)	Mécanique et ingénierie	CIS
BALBO	Flavien	PR2	Informatique	FAYOL
BASSEREAU	Jean-François	PR	Sciences et génie des matériaux	SMS
BATTON-HUBERT	Mireille	PR2	Sciences et génie de l'environnement	FAYOL
BEIGBEDER	Michel	MA(MDC)	Informatique	FAYOL
BLAYAC	Sylvain	MA(MDC)	Microélectronique	CMP
BOISSIER	Olivier	PR1	Informatique	FAYOL
BONNEFOY	Olivier	MA(MDC)	Génie des Procédés	SPIN
BORBELY	Andras	MR(DR2)	Sciences et génie des matériaux	SMS
BOUCHER	Xavier	PR2	Génie Industriel	FAYOL
BRODHAG	Christian	DR	Sciences et génie de l'environnement	FAYOL
BRUCHON	Julien	MA(MDC)	Mécanique et ingénierie	SMS
CAMEIRAO	Ana	MA(MDC)	Génie des Procédés	SPIN
CHRISTIEN	Frédéric	PR	Science et génie des matériaux	SMS
DAUZERE-PERES	Stéphane	PR1	Génie Industriel	CMP
DEBAYLE	Johan	MR	Sciences des Images et des Formes	SPIN
DEGEORGE	Jean-Michel	MA(MDC)	Génie industriel	Fayol
DELAFOSSE	David	PR0	Sciences et génie des matériaux	SMS
DELORME	Xavier	MA(MDC)	Génie industriel	FAYOL
DESRAYAUD	Christophe	PR1	Mécanique et ingénierie	SMS
DJENIZIAN	Thierry	PR	Science et génie des matériaux	CMP
DOUCE	Sandrine	PR2	Sciences de gestion	FAYOL
DRAPIER	Sylvain	PR1	Mécanique et ingénierie	SMS
FAUCHEU	Jenny	MA(MDC)	Sciences et génie des matériaux	SMS
FAVERGEON	Loïc	CR	Génie des Procédés	SPIN
FEILLET	Dominique	PR1	Génie Industriel	CMP
FOREST	Valérie	MA(MDC)	Génie des Procédés	CIS
FRACZKIEWICZ	Anna	DR	Sciences et génie des matériaux	SMS
GARCIA	Daniel	MR(DR2)	Sciences de la Terre	SPIN
GAVET	Yann	MA(MDC)	Sciences des Images et des Formes	SPIN
GERINGER	Jean	MA(MDC)	Sciences et génie des matériaux	CIS
GOEURIOT	Dominique	DR	Sciences et génie des matériaux	SMS
GONDRAN	Natacha	MA(MDC)	Sciences et génie de l'environnement	FAYOL
GONZALEZ FELIU	Jesus	MA(MDC)	Sciences économiques	FAYOL
GRAILLOT	Didier	DR	Sciences et génie de l'environnement	SPIN
GROSSEAU	Philippe	DR	Génie des Procédés	SPIN
GRUY	Frédéric	PR1	Génie des Procédés	SPIN
GUY	Bernard	DR	Sciences de la Terre	SPIN
HAN	Woo-Suck	MR	Mécanique et ingénierie	SMS
HERRI	Jean Michel	PR1	Génie des Procédés	SPIN
KERMOUCHE	Guillaume	PR2	Mécanique et Ingénierie	SMS
KLOCKER	Helmut	DR	Sciences et génie des matériaux	SMS
LAFOREST	Valérie	MR(DR2)	Sciences et génie de l'environnement	FAYOL
LERICHE	Rodolphe	CR	Mécanique et ingénierie	FAYOL
MALLIARAS	Georges	PR1	Microélectronique	CMP
MOLIMARD	Jérôme	PR2	Mécanique et ingénierie	CIS
MOUTTE	Jacques	CR	Génie des Procédés	SPIN
NEUBERT	Gilles			FAYOL
NIKOLOVSKI	Jean-Pierre	Ingénieur de recherche	Mécanique et ingénierie	CMP
NORTIER	Patrice	PR1	Génie des Procédés	SPIN
O CONNOR	Rodney Philip	MA(MDC)	Microélectronique	CMP
OWENS	Rosin	MA(MDC)	Microélectronique	CMP
PERES	Véronique	MR	Génie des Procédés	SPIN
PICARD	Gauthier	MA(MDC)	Informatique	FAYOL
PIJOLAT	Christophe	PR0	Génie des Procédés	SPIN
PINOLI	Jean Charles	PR0	Sciences des Images et des Formes	SPIN
POURCHEZ	Jérémy	MR	Génie des Procédés	CIS
ROUSSY	Agnès	MA(MDC)	Microélectronique	CMP
ROUSTANT	Olivier	MA(MDC)	Mathématiques appliquées	FAYOL
SANAUR	Sébastien	MA(MDC)	Microélectronique	CMP
STOLARZ	Jacques	CR	Sciences et génie des matériaux	SMS
TRIA	Assia	Ingénieur de recherche	Microélectronique	CMP
VALDIVIESO	François	PR2	Sciences et génie des matériaux	SMS
VIRICELLE	Jean Paul	DR	Génie des Procédés	SPIN
WOLSKI	Krzystof	DR	Sciences et génie des matériaux	SMS
XIE	Xiaolan	PR0	Génie industriel	CIS
YUGMA	Gallian	CR	Génie industriel	CMP

Mise à jour : 03/02/2017

Abstract

Smart Home, Ambient Intelligence and the Internet-of-Things involve a large number of connected objects, with heterogeneous computing and communication capabilities. The high-level functionalities offered by these systems are based on the services rendered by several of these objects in a joint manner; the coordination of their actions is therefore essential. In the current systems, this coordination is implemented via a centralized entity, the connected objects are then only used as simple effectors or sensors.

This thesis examines cooperation and coordination mechanisms, in a decentralized and autonomous way, between these objects. Based on a Multi-Agent System approach called Distributed Constraints Optimization (DCOP), these objects coordinate their actions to achieve one or more objectives corresponding to the user's requirements. In this context, we underline the importance of distributing the decisions to be taken by these various agents and we present several methods for choosing a satisfactory distribution against the characteristics of the targeted systems.

Finally, since these systems are highly dynamic by nature, we present several solutions to manage the changes that may occur, both in terms of the environment and the agents themselves. In particular, we are committed to making these systems resilient, so that they can continue to operate even in the event of the disappearance of several agents. Several autonomous system repair mechanisms, based on distributed decision replication and decision making, are presented and evaluated.

Acknowledgements

Contents

1 Introduction

Due to recent advance in technology and the decrease of the cost of embedded computing, both on the CPU and communication side, the ideas pioneered in academic works like *Pervasive Computing* and *Ubiquitous Computing* are currently materializing into real world solutions, which are generally referred to as the *Internet of Things*. In these systems many connected devices, typically equipped with computation capabilities, are used together and allow building high-level services, which can be aware of, and act on, the real world.

One specific use case for these systems is *Ambient Intelligence*, where this technology is used to facilitate the life of users by embedding computation, sensors and actuators in the environment. This area has been especially active in the past years, both in the industrial and the enthusiasts communities, with the emergence of many *Smart Home* solutions that seek to realize this idea in the home environment.

However, these systems are still in their infancy and their builders face many issues and challenges in order to reach the full potential of these new technical capabilities. In this thesis, we tackle some of these challenges and propose several approaches to overcome these issues.

1.1 Challenges and Approach

We envision an *Ambient Intelligence* setup as a distributed system where devices, which we consider as *agents*, have to cooperatively, autonomously and dynamically come up with a collective behavior that facilitates the life of the user. This solution will typically take the form of a *spontaneous configuration* of the environment, for example by setting, depending of various conditions, appropriate light, heat, humidity, etc. levels in the home. In any meaningful environment, many devices act on the same parameter (e.g. several light sources are used in the same area) and some coordination is required among them. Additionally, we want the users to be able to express their desired state of the environment, which the devices should reach autonomously.

Based on these desired characteristics for Ambient Intelligence systems, we identify several challenges, which we address in this thesis and that have been the subject of several contributions.

1.1.1 Modeling Goal-Oriented Smart Home Scenarios

The first question we want to answer in this thesis is how to model a Smart Home environment in a way that really matches the vision of Ambient Intelligence. One key element for that objective is that the user should not need to care about the inner workings of the system: detailed and manual configuration must be avoided in favor of the simple specification of goals. The devices should then autonomously decide how to reach these goals in the best possible way.

Another challenge, less explicit but also of paramount importance for achieving the Ambient Intelligence vision, is the issue of trust : How could we achieve a good user experience and a satisfactory quality of service while still preserving the privacy of the user? Many current approaches are based on remote centralized reasoning, where the data are sent to a cloud-based reasoning module, which make decisions. Such architectures are problematic from a privacy perspective, and also introduce points of failure in the system: if the connection is lost or the gateway fails, the whole system stops operating.

We advocate that the operation of such intelligent environments cannot be traded against user privacy and that all decisions and data should be kept locally in the home, in a way that ensures both privacy and robustness.

To answer this challenge, we decide to model coordination in an Ambient Intelligence environment as an optimization problem and to use the **Distributed Constraint Optimization Problem** (**DCOP**) framework to address this problem in a fully decentralized way. We call the resulting model the **Smart Environment Configuration Problem** (**SECP**). This model has been presented at **IJCAI** [1] [128] and **JFSMA** [2] [122] in 2016.

1.1.2 Installing Decentralized Coordination In the Real World

Following the first challenge, an important question to answer is how to install such distributed model on the real devices that the smart environment is made of.

Indeed, most current research on the **DCOP** framework is based on assumptions that do not hold when applying the model to real world situations like Ambient Intelligence. When solving an optimization problem distributively, the decision making process is implemented through computations that perform the optimization process and must physically run on some hardware substrate. Traditional **DCOP** approaches consider that each variable in the problem, and the corresponding decision making process, is bound to exactly one agent. This agent is responsible for selecting the value for this variable and make its decision based on messages exchanged with other agents responsible for a variable that shares a binary constraint with it. When a problem contains non-binary constraints, it is usually *binarized* (i.e. mapped to an equivalent problem that only have binary constraints, by adding variables) and agents are added to obtain exactly one agent for each variable.

However, in real world problems like the SECP, constraints are often non-binary and the set of agents is given by the problem definition; in our case, the devices in the systems are the physical artifacts that embody our agents and the system can only run on available devices.

1. **International Joint Conference on Artificial Intelligence (IJCAI)**
2. **Journées Francophones sur les Systèmes Multi-Agents (JFSMA)**

As a consequence, we argue that in order to apply the **DCOP** framework to real world problems, these decision making computations must be distributed on the agents/devices. This distribution must take into account the characteristics of the target environment, which encompass, for the SECP, the limited capabilities of the devices and the constrained communication.

We propose a definition of optimality for such distribution, both for SECP and for more generic IoT system, and develop several approaches for computing these distributions, both optimal and heuristic. This work has been initially published at **OptMAS** [3] [126] and **JFSMA** [123] in 2017.

1.1.3 Providing Resilience in Decentralized Decision Making

The goal of Ambient Intelligence systems is to facilitate the life of its users, who should ideally almost forget that the system is actually working in background and simply enjoy the services it provides. Therefore, an important challenge of such systems is ensuring reliability and resilience. However, a smart environment is also a dynamic and open system: the characteristics of the problem may change at runtime and devices may join or leave the system at any moment. Thus, the question is how to ensure that the system keeps providing the services required by the users, with the equivalent quality of service and quality of experience, whilst the infrastructure is changing.

For this purpose, we define the concept of k-resilience and propose several solution methods to achieve this, by using distributed replication and repair techniques. These approaches have been presented at **OptMAS** [121] in 2018, **AAMAS** [4] [124] and **JFSMA** [127] in 2019.

1.1.4 Designing and Developing Decentralized Coordination Mechanisms

While working on the aforementioned topics, we realized that the software tools and libraries used by the **DCOP** community were not suitable for our study as they do not target real world usage of **DCOP** and do not consider the problem of decisions distribution, as they are generally based the classical assumptions. Besides, most of these libraries are not maintained and no unified repository of common **DCOP** algorithms could be found.

Therefore, we decided to develop our own **DCOP** library, pyDCOP [5], specifically designed for studying the use of **DCOP** in **IoT** systems. It can also be used for the general study and design of **DCOP** algorithms. With its extensive documentation and the inclusion of many algorithms implementations, we hope it will foster research in these areas.

pyDCOP has been published in open source in 2017 and was presented to the community at **OptMAS** [120] in 2019. pyDCOP has also been used for lectures and tutorials at **EASSS** [6] in 2018, at **AAMAS** in 2018 and 2019 and at **PFIA** [7] in 2019. A physical demonstration of a distributed resilient decision making system, implemented with pyDCOP, has also been presented at **JFSMA** in 2019.

3. **Optimization in Multiagent Systems (OptMAS)**, an **AAMAS** workshop
4. **International Conference on Autonomous Agents and Multiagent Systems (AAMAS)**
5. https://github.com/Orange-OpenSource/pyDcop
6. **European Agent Systems Summer School (EASSS)**
7. **Plate-Forme Intelligence Artificielle (PFIA)**

1.2 Overview

Chapter 2 introduces relevant related works, both in Ambient Intelligence and Distributed Constraints Reasoning. We present the current state of IoT and Ambient Intelligence, the implementation model commonly used and its limits. We also briefly introduce the field of Multi-Agent Systems and focus on the domain our work is based on: Distributed Constraints Optimization. We expound major solutions methods and variants and tackle academic works that address Ambient Intelligence challenges using Distributed reasoning.

Chapter 3 presents our model for distributed coordination in smart environment and explains how we can map it to a DCOP, which solving process yields an environment configuration that matches users preferences.

Chapter 4 focuses on the distribution of decisions on a physical infrastructure made of the devices available in the smart environment. We also extend our definition of distribution to encompass more general systems like IoT. In both case several solution methods for distribution are presented and evaluated.

Chapter 5 explains how smart environments must be conceived as dynamic and details consequences of these dynamics, by classifying them into two categories: the computation level and the infrastructure level, which we focus on. Several approaches are introduced to deal with the arrival and departure of agents in the system.

Chapter 6 presents our software library for DCOP study, pyDCOP. Major related libraries are introduced, detailing why we feel that a new one is needed. The architecture and usage of pyDCOP are presented, along with one example use in a demonstration.

Chapter 7 concludes this thesis by summarizing results and contributions and identifying directions for future research.

2 State of the Art on Ambient Intelligence and Distributed Reasoning

In this chapter, we propose a short review of the state of the art on Ambient Intelligence, with its current industrial foundation –the Internet of Things–, and a particular branch of Multi-Agent Systems, Distributed Reasoning. We then concentrate the review on Distributed Constraint Optimization, as our work makes use of this approach to implement Distributed Ambient Intelligence.

2.1 Embedding Technology in Everyday Life

Embedding technology in everyday life is a long trend that started decades ago when personal computers became common in houses and is today still very active with more and more connected devices. This trend has been studied, and anticipated, by several research and industrial communities, among which we focus on Ambient Intelligence and Internet of Things.

2.1.1 Ambient Intelligence

Ambient Intelligence (AmI) [31] refers to physical environments that adapt themselves to best fit the needs and expectations of their human inhabitants and users; the overarching goal here is to facilitate the life of users by using technology. To achieve this goal these environments are fitted with connected devices, sensors and actuators, which enable the system to be aware of the context and react to changes that may arise both from users and from external elements (e.g. temperature, light, humidity, time).

The **AmI** vision focuses on the user experience: the devices, and the technology in general, must blend into the surrounding up to the point that the users does not feel their presence and the environment itself, with its automatic adaptation, is the only visible user interface behind which all technical details are hidden. The system must work pervasively, be non-intrusive and transparently assist the user in his everyday tasks.

The **AmI** is a multidisciplinary paradigm which dates from the late 1990s and builds upon *Pervasive Computing* [130] and *Ubiquitous Computing* [146], as it requires the technical infrastructure envisioned by these paradigms: computing power embedded in most devices and communication network allowing the seamless coordination of these devices. To achieve its objective, **AmI**

traditionally also makes use of many **Artificial Intelligence (AI)** methods especially when it come to coordinate several dedicated devices and services so as to provide seamless cross-concern added value functionalities.

The **AmI** concepts can be applied in the public space, with smart street lighting for example, in the work spaces and in the domestic space. For instance, [129] lists the following, non-exhaustive, applications areas for **AmI**: home, health care, business, commerce, leisure and tourism. When applied to the home, **AmI** can be related to the idea of *Smart Home* and is also called **Smart Home Environment (SHE)**. The area of smart homes can also be considered as a specific use case of ubiquitous computing that integrates **AmI** and automatic control into living spaces According to [5], this integration objectives include comfort, entertainment, healthcare, assisted living security, and energy efficiency.

2.1.2 The Rise of the Internet of Things

The **Internet of Things (IoT)** vision is a big trend today, both from a marketing, technical and research point of view. The term **IoT**, coined by Kevin Asthon in 1999, describes a system where most physical objects are connected to the Internet and can be queried and controlled remotely. The basic idea of this vision is that all objects, these so called *Things*, are provided with communication capability and have some processing power.

This idea of embedding computational and communication capabilities into everyday objects, and thus considering our environment as a system made of interconnected devices, is not entirely new; it was already studied in the 1970s and was then called *Pervasive Computing* or *Ubiquitous Computing*. But while these visions were more academic, the current IoT trend also involves the industry in many, if not all, sectors.

Analysts forecast that more than 20 billion connected objects [42] will be in use by 2020. One major drive for this incredible growth is the availability of cheap hardware: progress in CPU manufacturing and communication technologies allows producing very cheap chipsets, which can be embedded in almost any object, where is was before technically and economically not feasible.

It is envisioned that this interconnection of devices will allow the development of new functionalities and services in all domains; manufacturing, healthcare, city management and also in the consumer market with devices in the personal sphere like connected watches, vocal assistants and smart home products. Another hope for IoT is that it will allow to break down the silos between different domains; data and functionalities coming from devices originating from different application domains can be combined and used together to create new services and insights.

2.1.3 Implementing Ambient Intelligence

This rise of **IoT** is the technical foundation on which the promises of Pervasive Computing and **AmI** can be built; the various communication mechanisms, the cheap price of chipsets and the small physical size of these technical blocks have lead to huge number of connected "*smart*" devices that one can see everyday. Using these devices, and the overall connectivity allowed by mobile broadband, as a technical infrastructure, one should be able to implement the *pervasive* and *ambient* aspects of computation that was envisioned by early academic works. Notice that, while

AmI has relationships [7] with many areas in computer science, electronics and communication technologies, in this work we study its *intelligence* and *reasoning* dimensions.

Actually, the idea of a consumer-ready Smart Home is far from new, the idea was previously known as *Home Automation* and was already pictured in the movie *My Oncle* from Jacques Tati in 1958! Following the **IoT** trend, Smart Home systems have been very popular during the last years, both in the academic domain and the mass consumer market, with the availability of products from many different companies ranging from Telecommunication Operators (like Homelive [1] and Maison Connectée [2] from Orange in France, Qivicon [3] from Deutsch Telecom or my-digital-life [4] from ATT), big Internet companies (like Apple with Homekit [5]), equipment manufacturer (like Smartthings [6] from Samsung) and many startups and open-source projects like jeedom [7], domoticz [8], openhab [9], to only cite a few.

However the current production-level systems are generally far from being actually smart and are simply more or less advanced *remote-control* and *automation* systems; these products are quite far from the initial expectations from **AmI**, and much of the *Intelligence* envisioned is still in the domain of academic research. Data from the sensors, like temperature for example, can be read remotely and actions of the devices can be triggered, typically through a mobile application. Additionally these systems generally provide a simple condition-action mechanism, where actions are executed when a predefined (but configurable) set of conditions, based on sensor values, are met. Most of the time, these automatic behaviors must be manually configured by the end-user(s), which is complex and error prone. Moreover, these systems are still currently plagued by interoperability issues, at all levels of the communication stack: physical and data layers (various radio technologies), transport layer (IP, 6LowPan, etc.) and application layer (CoAP, mqtt). While a standard war is raging today, many companies choose to use their own proprietary solutions, and no winner can be identified in the foreseeable future.

When it comes to implementation, as noted in [109], we can distinguish two main architecture styles for **AmI** and **SHE**: centralized and distributed solutions. This distinction could also be applied to other IoT-based solutions and, while we focus on **AmI** in this work, we believe that the following could apply in many other use cases. It can be noted that the separation between the centralized and distributed approaches is not strict: depending on the desired characteristics of the system, one can design partially centralized (or partially distributed) systems.

2.1.3.1 Centralized Ambient Intelligence

This first approach is based on a central reasoner; data is collected from all sensors in a single place, where decisions are made. Based on the outcome, orders are then sent to actuators, which simply apply what has been decided, without any autonomy nor decision power. This decision

1. http://homelive.orange.fr
2. https://boutique.orange.fr/maison/domotique/
3. http://qivicon.com
4. https://my-digitallife.att.com
5. https://developer.apple.com/homekit
6. https://www.samsung.com/us/smart-home/smartthings/
7. https://www.jeedom.com
8. https://www.domoticz.com
9. https://www.openhab.org

center is thus omniscient; it has access to all sensor data and all the (possibly conflicting) goals of the system. It can consider all parameters, which theoretically allows it to make optimal decisions for the problem.

This centralized approach is generally considered to be easier to implement: the central reasoner can use many well-known and proven decision-making mechanisms like rule engines, solvers, or optimizers. However, centralizing the decision-making process also introduces some limitations and problems of its own. The most obvious drawback of centralization is that it introduce a **Single Point Of Failure (SPOF)** in the system: if the central reasoner fails or communication is lost, the system ceases to operate entirely. Even when the reasoner works correctly, it can become a bottleneck, especially from a communication point of view, as all other devices, sensors and actuators, must be able communicate with it at any time. The fact that this central brain reasons on all parameters can also become a burden when the global systems get larger: when the number of devices increases the problem to solve becomes exponentially larger and might become very difficult and even impossible to work on in a centralized manner, at least in a reasonable time. While this would probably not happen in home-sized systems, this aspect certainly limits the applicability of the centralized approach for the public space. Finally the centralized approach is also probably more difficult to apply in dynamic environments: the insertion and deletion of any devices must be detected by the reasoner.

This centralized approach is currently used by many hobbyist (generally non-commercial) smart home systems, where a *smart home box* acts as a central brain and coordinates all actions thanks to a very delicate configuration by the user.

This idea of a *smart home box* has been the origin of many academic and industrial works. For example, the **Open Service Gateway Intiative (OSGi)** [92], founded in 1999, has designed a middleware platform initially specifically for residential gateways. It has latter extended its target use cases to address software modularity in general but still has a strong presence and focus on **IoT** and smart home scenarios.

OSGi has been the basis of numerous works in the *Pervasive Computing* and **AmI** research communities, like [71, 110]. It is used today in several commercial products, like Qivicon's [10] *Home Base* or or Bosh's [11] IoT gateway platform (formerly Prosyst [12]). In the hobbyist market, openHAB [13] is an **OSGi**-based local and centralized home automation solution, with a strong focus on privacy.

2.1.3.2 Partially Centralized Ambient Intelligence

While the fully centralized, and local, approach is the most common in hobbyist systems, most commercial offers are based on a hybrid architecture, where the sensor's data is collected by a local device, which forward most, if not all, data to a cloud service. This device, which can be seen as an evolution of the Smart Home box, is often called a *Smart Home Gateway*. This architecture is for example used by all commercial products mentioned previously (Orange, Samsung, qivicon, etc.)

10. https://www.qivicon.com
11. https://www.bosch-si.com/iot-platform/iot-platform/gateway/software.html
12. https://en.wikipedia.org/wiki/ProSyst
13. https://www.openhab.org

and has also been the suject of academic works like [149] where Xiaojing Ye and Junwei Huang argue that is allows for better extensibility and interconnection capabilities.

Reasoning is then shared, at various degrees, between the cloud service and the local gateway. In some cases, the gateway simply acts as a relay and all the decisions are made in the cloud, leaving the system entirely broken if the Internet connection is down. In some other cases, the box implements basic functions and can act as a fallback when the connection to the cloud is lost. In the best case, the reasoning and decision is performed locally, and the cloud is only used for computation-intensive tasks, like for example using machine learning algorithms, the results of whose is then transferred back in the local box where it can be used for local decisions.

The cloud service is also often used to provide interoperability with services that do not offer a local interface that could be used in the home. This could be because these services are not physically available in the home (let's say for example a car localization service) or because their designer simply decided to only provide a cloud-based interface (like for instance the Netatmo's weather station indoor module [14], which measures the temperature in the home but provide no local network interface).

This two-tiers approach is the most common today, but one could think of other kind of partially centralized systems. For example, one could build hierarchical systems, where several "local" reasoners work each on a subset of the space. However, this also introduces additional complexity, like the need of coordinating these reasoners. This approach is not used, as far as we know, in any current Smart Home solution although it has been studied in **AmI** works for public space, like [2] which uses a hierarchical holonic agent organization for traffic signals control.

2.1.3.3 Distributed Ambient Intelligence

The second approach is to consider the **AmI** system as a network of interconnected devices, which can interact with each other without relying on any central element. In this approach, devices coordinate directly to reach the goal(s) set to the system. This leads devices to concentrate on smaller, more focused problems (which, for example, may depend on the functions of the device) and introduces some notion of *neighborhood*. Devices communicate and coordinate with other devices in their neighborhood, working on what can be seen as local goals. As goals may also depend on each other (one outcome may influence, or even contradict, another) these neighborhoods are interlaced and the whole system can be seen as a graph where devices are vertices and communicate along the edges of the graph. Although not specific to **AmI**, mesh networks are an example of such approach, in which nodes cooperate and coordinate to route network data, which enables dynamic self-organization and self-configuration.

One interesting property of this architecture is that the system can keep working, even if in a degraded mode, in the case of a connection loss or a device failure.

Moreover, as the devices work on smaller subset of the whole problem, and no central point needs to know all the parameters, the resulting system may be able handle large scale systems better than the centralized approach.

However, building such distributed **AmI** requires coordination and distributed decision making

14. https://www.netatmo.com/en-us/weather/weatherstation

techniques, which are generally considered to be more complex than centralized reasoning methods. For this reason, many academic works on distributed **AmI** are based on the multi-agent paradigm, which will be presented and discussed in the next section.

2.2 Multi-Agent Systems

In this section we briefly introduce the **Multi-Agent Systems (MAS)** framework and explain the reason we consider it is a good candidate for implementing **AmI**. Then we present a few selected academic works tackling smart environment and **AmI** issues by using this framework.

2.2.1 A Quick Overview of Multi-Agent Systems

MAS have emerged as one important areas of research in **AI** and computer science in the 1990s, and can be seen as the evolution of **Distributed Artificial intelligence (DAI)**. They are considered to bring significant advantages when it comes to designing systems that are **complex**, **distributed** and **dynamic**.

As noted by Shoham in [133] several different, and mutually inconsistent, definitions of **MAS** coexist. An easy, but loose, way to define them is to say that a multi-agent system is one composed of several intelligent entities (a.k.a. *agents*), which **interact** with one-another. Like in the classical agent-view of **AI**, agents in a **MAS** are **autonomous**, are situated in an **environment**, which they can sense and act on, and are generally **reactive** and / or **goal-oriented** (pro-active). The main addition of the **MAS** paradigm is that agents have social capabilities. They are collectively capable of reaching goals that would be difficult to achieve by a single agent or a monolithic system.

For that purpose, in a **MAS** an agent's social ability generally includes one or several of the following characteristics:

- *cooperation*: working together,
- *collaboration*: managing the inter-dependencies between activities,
- *negotiation*: reaching agreement.

Additionally, in many works, **MAS** are considered to be self-organized [32] autonomous systems, meaning that agents should not require external intervention to accomplish the task assigned to the system.

2.2.1.1 MAS Characteristics

Given the lack of an accepted standard definition for **MAS**, it is difficult to come up with a comprehensive list of the characteristics that distinguish it from a single-agent system. Thus, we provide here a list of the characteristics that everyone in the domain should be able to accept and that we, probably with a one-sided view, consider as essential for a **MAS**.

MAS Environment

Like single agents, agents of a **MAS** are situated in an environment. But while most single agents are designed with a static environment in mind, in a **MAS** the mere presence of several agents makes the **dynamic nature of the environment** more obvious and practically difficult

to ignore. In a **MAS** each agent is supposed to be autonomous and organization among them may emerge spontaneously (i.e. **MAS** are self-organized systems). This makes these systems very modular and should help in handling the dynamic nature of the environment and/or the problem to be solved.

Interaction

Agents in a **MAS** coordinate, which generally requires some kinds of communication with one another with the notable exception of systems based on *stigmergy*, where communication is indirect and happens through the environment. In other cases, when agents communicate, an interaction structure, called *interaction protocol* is required to define the pattern(s) of messages exchange used to implement such coordination.

Organization

The social abilities of agents in a **MAS** lead naturally to consider the whole system as an *agent society* and many works focus on the organizational models [52] of these societies using various coordination patterns: teams, groups, roles, norms, etc.

Robustness

A **MAS** should be able to tolerate agents failures and keep working toward the shared goal(s). This quality stems from the fact that control and responsibility is shared among agents.

Scalability

As a **MAS** already encompasses many agents, which work through coordination, it can more easily integrate new agents and potentially solve bigger problem. Additionally the overall goal is general subdivided in several overlapping smaller local, and thus more tractable, goals, which makes adding new goals and increasing the problem complexity easier.

2.2.1.2 Challenges Addressed by MAS

The **MAS** framework can be used on many problems, we try here to classify these in four major domains:

Problem Solving

Problem solving and more specifically *distributed* problem solving, is a major application for **MAS**, which can be used either when the problem is distributed by nature or because distributing the problem is more efficient for solving it. Additionally, distributed problem solving can be implemented so that to enhance the robustness and the flexibility of the system, compared to a centralized approach. This can be applied to many kind of problems, like for example resources allocation or distributed planning.

Simulation

MAS are widely used in simulation in various domains like social science, biology and even computer graphics. This approach is particularly relevant when the simulated system is complex, dynamic and stochastic by nature and it is not possible to build an analytical model for it. In such cases, one may build simple interaction and behavior models for agents, and observe how they interact when put in the same environment.

Cyber-physical Systems

These systems are made of many interacting elements and mix both physical and software

components. This is for example the case in Smart Grids, Autonomous cars, or Collective Robotics systems. These systems usually require coordination and cooperation to reach the collective goal(s). The **MAS** framework provides concepts and solutions to serve as the foundation of such systems.

Distributed and Decentralized Software Systems

More generally, all distributed systems (like parallel computation, distributed databases, cluster hosting, communication networks, etc.) require many distributed software components to communicate and coordinate. In this kind of systems, the **MAS** approach is often used to provide greater flexibility, and facilitate the integration of autonomous software services, especially when the set of services is dynamic and possibly not fully known *a priori*.

2.2.1.3 Some MAS Approaches

Many research topics have emerged in the **MAS** community, we only cite here the approaches that focus on the reasoning dimension as this is the domain the work we present in this document is based on. Reasoning approaches in a **MAS** can be roughly categorized into four domains.

Decision Theory

Decision theory aims at modeling uncertainty at all levels in the system explicitly, and allows capturing very complex scenarios, at the cost of a very high complexity [112]. The **Markov Decision Process (MPD)** framework, popular modeling single-agent decisions, have been extended by **MAS** researchers to the decentralized use-cases with **Decentralized Markov Decision Process (Dec-MDP)** and **Decentralized Partially Observable Markov Decision Process (Dec-POMDP)** [10]. These extensions offer a rich framework for modeling cooperative sequential decision making under uncertainty but exhibit a **NEXP**-complete worst case complexity.

Game Theory

This theoretical framework, which is closely related to Decision Theory, is also considered to be an important branch of **MAS**, which studies decision making by agents and focus on the interactions of the agent's decisions and their influence on the decision making process [12, 133].

BDI Agent Model

This model is inspired by Michael Bratman's model of human practical reasoning and stands for *Belief-Desire-Intention* [18, 113]. The idea is that agents, based on the information they possess on the environment and other agents (their *Beliefs*), commit to perform some concrete actions (expressed as *Intentions*) in order to ultimately reach their overarching goals (the *Desires*).

Distributed Constraint Reasoning

This framework is another **MAS** approach that uses constraint reasoning [28] (satisfaction or optimization) to implement coordination among agents in **MAS** [38, 153, 154]. This approach will be discussed further in Section 2.3.2.

2.2.2 Application of Multi-Agent Systems to Ambient Intelligence

In order to really make the **AmI** vision a reality, one needs to implements many functions which are studied as part of **AI**: context management, semantics, coordination, self-adaptation, planning, reasoning, learning and natural language processing are some examples of such functions; see [5, 27, 129] for a (non-exhaustive) list of **AI** approaches and algorithms that have been used on **AmI** and **SHE** use cases. As such, many researchers in **AI** have studied the **AmI** use cases.

Being a distributed problem, **AmI** naturally leads to **DAI** and the **MAS** research domain, where the system is composed of multiple autonomous intelligent agents. These agents interact and are able to collectively tackle problems that would be difficult or even impossible to address with a single entity.

As previously noted, **MAS** are considered to be a suitable paradigm for complex adaptive systems, especially when they are distributed and dynamic. Parunak in [97], for example, argues that an agent approach is appropriate for applications that are modular, naturally decentralized, changeable, ill-structured, and complex.

While, as far as we know, no **MAS**-based commercial application exists yet for **AmI** and **SHE**, it has been studied in many academic papers from several research communities.

For example, in [148], Wu et al. propose an architecture for **SHE**, based on **Service Oriented Architecture (SOA)** and a **MAS** that also includes Mobile Agents [15].

In [138], Sun et al. use **Belief, Desire and Intention (BDI)** agents [113]. The system is composed of agents of four different types: sensing, action, decision and database. Each agent maintains a set of beliefs about the environment and, given a user goal, selects a set of suitable desires and intentions (i.e. action plans) to decide its individual behavior. Multi-agent group behavior is based on user-defined regulation policy, which are used to generate agents collaboration protocols. Various policies can be used to give the system specific characteristics: for example, it could favor either quality of service, response time or low power consumption. In this work, agents are not tied to specific devices, they are hosted on a distributed agent execution platform, implemented using **Java Agent Development Environment (JADE)** [16] [9], running on a group of heterogeneous hosts, which may include smart devices, sensors and any other devices with computing capability.

Rodríguez et al. also proposed to use a **MAS** approach for information fusion and management in a residential environment [116]. Their system takes advantage of the modularity of **MAS** to allow the inclusion of organizational concepts, including rules, norms and social structures and ease the dynamic integration of new information fusion techniques.

In his PhD thesis [46], Guivarch uses an **Adaptative Multi-Agent Systems (AMAS)** approach to take into account the the context's dynamic. In his system, each agent learns its appropriate behaviour, based on the context and information transmitted by other agents. He argues that this approach allows providing solutions to problems that are not well specified and for which it is not possible to identify a solution *a priori*.

Mazac, Armetta, and Hassas also study [83] the use of **MAS** for **AmI**, more specifically for

15. Mobile agents are an extension of the classical AI agent paradigm where agents are *mobile*; i.e. they can migrate from one computer to another. When moving, a mobile agent transport both its logic and its state.
16. https://jade.tilab.com

bootstrapping a constructivist learning process in a continuous environment. They argue that a self-organising **MAS** can be used for the construction of initial patterns, required to bootstrap the learning process, and successfully recognize significative events, even without any *a priori* knowledge.

Some work also use a **MAS** approach to **AmI** and **SHE** but run all the agents in a single physical computing platform. In that case, the solutions is centralized, from a deployment point-of-view. This is for instance the case in [140] where Valero et al. use a **MAS** approach to facilitate the management of the multiple occupancy in smart living spaces and motivate the use of the **MAS** framework by the use of the organization, norms and roles concepts provided by the Magentix **MAS** platform,

In [95], Palanca et al. design a goal-oriented **SHE** using a **MAS** approach. Instead of telling the system what should be done, users express high level goals which are fulfilled by composing a set of services offered by agents in the system. Agents are **BDI**-inspired and posses a set of beliefs, goals and small-scale plans. While the goal-oriented approach is very interesting, in this work the overall plan (i.e the cross-agent actions needed to fulfill the user goals) selection is implemented by the agent framework, mode precisely by a *deliberation engine* and not collaboratively by the agents themselves.

In [108], Piette et al. use a **MAS** approach for the dynamic deployment of the software components required for a distributed application in an **AmI** environment. They advocate that the use of **MAS** allows to preserve privacy at architecture and organisation levels and also enhance the robustness of the solution.

Other works, like [142] from Vallée et al., propose to combine **SOA** and **MAS** to implement **AmI**. In this view, low-level device functionalities are represented as **SOA** services and *service composition* is used to aggregate them into higher-level services. Considering everyday environments as open and dynamic systems where resources, context and activities change continuously, the authors argue that a **MAS** approach is ideal to provide a dynamic service composition infrastructure.

Some other works use a **Distributed Constraint Optimization Problem (DCOP)**-based approach to implement coordination among agents in a **SHE**. These works, and the **DCOP** framework, are discussed respectively in Sections 2.5.2 and 2.3.2

2.3 Distributed Constraint Reasoning

In this section, we introduce the *Distributed Constraint Reasoning* framework, starting with its classical centralized approach and moving on to its distributed counterpart. We focus particularly on **Distributed Constraint Optimization Problem (DCOP)**, which is the main solution used in this work, including the common assumptions used by most researches in the domain and the main extensions to that framework.

2.3.1 Constraint Reasoning

One popular approach for reasoning and decision making is constraint processing. We only give here a simple definition of a constraint reasoning problem, for more details on this topic, we redirect

the reader to the Dechter's book [28].

Definition 1 (CRP). *A **Constraint Reasoning Problem (CRP)** is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{F} \rangle$ where*

- $\mathcal{X} = \{x_1, \ldots, x_i\}$ *is a set of discrete variables,*
- $\mathcal{D} = \{\mathcal{D}_{x_1}, \ldots, \mathcal{D}_{x_i}\}$ *is a set of finite domains, with variable x_i taking it its value in $\mathcal{D}_{x_i} = \{v_1, \ldots, v_k\}$,*
- $\mathcal{F} = \{f_1, \ldots, f_j\}$ *is a set of constraints on the values that the variables might take on simultaneously.*

Each constraint f_j is a function. The set of variables involved in this function is called the *scope* of a the constraint and denoted $S_{f_j} \subseteq \mathcal{X}$. These constraints be can hard or soft.

A **soft constraint** indicates preferences: f_j is an utility function which assigns a real valued reward (also called utility) for each possible combination of values of the variables in the scope S_{f_j} of the constraint. Formally, $f_j : \prod_{x_p \in S_{f_j}} \mathcal{D}_{x_p} \mapsto \mathbb{R}$, where $\prod$ is the Cartesian product. We denote r the arity of the constraint: $r = \|S_{f_j}\|$.

On the other hand, a **hard constraint** only allows some combinations of values, other assignments being explicitly prohibited. In that case, f_j is a relation between the domains of the variables in it scope S_{f_j} and can be represented extensively as a list of allowed assignments for these variables. Formally, $f_j \subseteq \prod_{x_p \in S_{f_j}} \mathcal{D}_{x_p}$.

An **assignment** of values to variables in $\mathcal{X}$ is said to be **complete** if every variable is assigned, otherwise it's a **partial assignment**.

The problem is called a **Constraint Satisfaction Problem (CSP)** if all the constraints are hard. A solution to a **CSP** is a complete assignment that satisfies all the constraints in $\mathcal{F}$. Such assignment is called **a consistent assignment**.

If the problem involves soft constraints the problem is called a **Constraint Optimization Problem (COP)** and a solution is a complete assignment that optimizes a global function defined as an aggregation of the utility functions of the constraints. This optimization can be either a maximization or a minimization. A weighted sum is generally used as a simple aggregation method. It should be noted that a **CSP** can always be transformed into as a **COP** by encoding hard constraints as soft constraints whose functions assign 0 to allowed combinations of values and $-\infty$ (for maximization) or $+\infty$ (for minimization) for other combination.

Constraint reasoning has been successfully used in many situations, both in operational research and AI. Current constraint reasoning techniques allow tackling complex problems, and many problems can be formulated as constraints satisfaction or optimization problems, like resources allocation, planning and scheduling.

Many extensions of **CRP** have been studied, like Dynamic Constraint Reasoning, Temporal **CSP**, Constraint Logic Programming, etc. One extension we are particularly interested in here, Distributed Constraint Reasoning, is presented in 2.3.2. It should be noted that some approaches (e.g. *Linear Programming*), which we do not discuss here, also consider infinite domains and non discrete variables.

Map coloring problems are a common example of a **CSP**, often used for benchmarks. In these problems, we consider a graph where each vertex is a variable representing a region of a map that

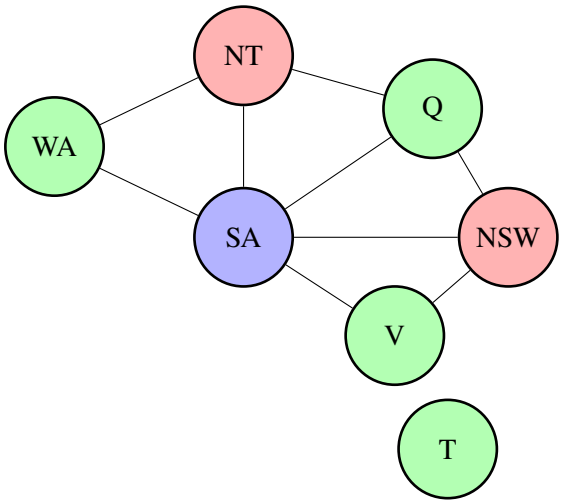


Figure 2.1 – A sample map-coloring problem on Australia

must be *colored*. Edges represent adjacency of two regions and the set of available colors is the domain of the variables. The goal is to find an assignment where any two adjacent regions/variables take different colors. Graph coloring problems will be later used in this document, to benchmark both our distribution approaches (Section 4.6) and our self-repair methods (Section 5.7).

Example 1. *Figure 2.1 represents a very simple map-coloring problem applied on Australia. Each region of Australia is represented by a variable, whose domain is the color set* $\{red,\ green,\ blue\}$. *The goal is to color the map so that no two adjacent regions are of the same color.*

This setting can of course be extended to general graphs, and is then called *graph-coloring*. When, instead of using hard constraints, we define a cost for any combination of colors for two adjacent vertices, the problem is called a *weighted graph coloring problem* or *soft graph coloring problem* and is also often used for benchmarking constraint optimization solution methods.

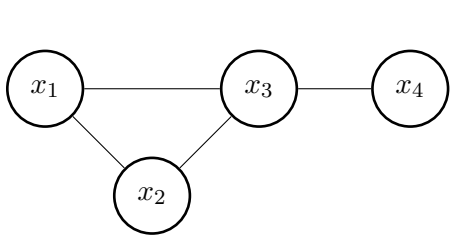
2.3.1.1 Graphical Representation

CRP are often represented as graphs, like we just did with a simple map coloring problem on Figure 2.1.

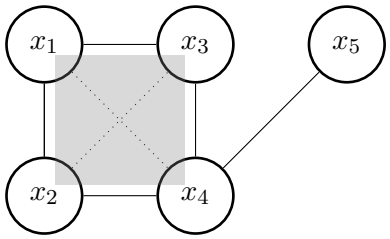
The standard graphical model for representing a **CSP** or a **COP** is a **constraint graph** (a.k.a. **constraint network**): an undirected graph $G = \langle \mathcal{X}, E_G \rangle$ where each vertex $x \in \mathcal{X}$ represents one variable and edges represent the binary constraints between the variables represented by the vertices at the ends of the edge: $E_G = \{(x_i, x_j) \mid \exists f_k \in \mathcal{F},\ \{x_i, x_j\} \subset S_{f_k}\}$.

This representation, used in Figure 2.2a, is efficient and commonly adopted, as most problems are usually formulated only with binary constraints (see Section 2.3.2.2 for a discussion on this point), but is not convenient when the problem contains non-binary constraints. A non-binary constraint can be seen as a clique among the variables involved in the constraint and some authors represent them as areas drawn on the graph, as depicted on Figure 2.2b. However the result is not readable but for the simplest constraint networks.

Constraint networks with non-binary constraints can also be represented as constraints hypergraphs, with hyperedges corresponding to constraints. Two vertices are in the same hyperedge if they are



(a) Simple constraint graph with only binary constraints



(b) Constraint graph representation for *n-ary* constraints

Figure 2.2 – Standard constraint graph representations

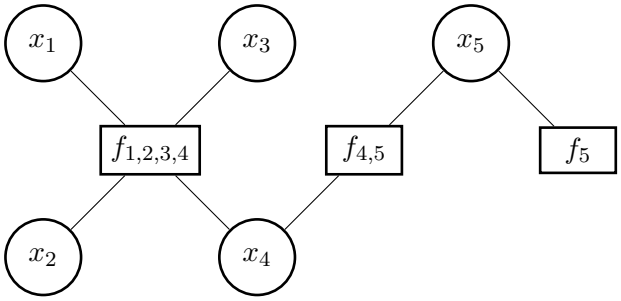


Figure 2.3 – Factor graph representation

involved in the same constraint. However, while this representation can be used by algorithms, hypergraphs cannot be easily visualized and must be transformed into classical graphs for that purpose.

Another solution is to use a **Factor Graph (FG)** [65, 66], which is an undirected bipartite graph $F = \langle \mathcal{X}, \mathcal{F}, E_F \rangle$ where the vertices $x_i \in \mathcal{X}$ represent variables and vertices $f_i \in \mathcal{F}$ represent constraints (called factors). Variable vertices are usually depicted with a circle and constraints with a rectangle, as illustrated on Figure 2.3. An undirected edge exists in the factor graph between a variable and a constraint vertices if the variable is in the scope of that constraint: $E_F = \{(x_i, f_k) \mid x_i \in S_{f_k}\}$.

All these graphical representations are not only useful for visualizing the **CRP**, they also often serve as the structure on top of which many constraint reasoning algorithms can operate. Traditional constraint satisfaction algorithms, for example, use arc consistency on a constraint graph, while other algorithms originating from signal processing and statistical learning use local message passing over these graphs. This is even more preeminent in the distributed variant of constraint reasoning and will be discussed in 2.3.2.1.

2.3.2 Distributed Constraint Reasoning

While solution methods for standard **CSP** and **COP** are usually centralized, an extension of these topics has emerged in the **MAS** research eco-system, where the constraint reasoning process is distributed among agents. Each agent has control over some of the variables and agents interact to find a solution to the problem.

Distributed Constraint Satisfaction Problem (DCSP) is the distributed counterpart of **CSP** and only considers hard constraints. It was initially proposed by Yokoo, Ishida, Durfee, and Kuwabara

in [154] to formalize distributed problem solving. It was latter extended, and superseded in most works, by the **DCOP** framework.

Definition 2 (DCOP)**.** *A discrete **Distributed Constraint Optimization Problem (DCOP)** is formally represented by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$, where*

- $\mathcal{A} = \{a_1, \ldots, a_{|A|}\}$ *is a set of agents,*
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ *are discrete variables, owned by the agents,*
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ *is a set of finite domains, such that variable x_i takes values in $\mathcal{D}_i = \{v_1^i, \ldots, v_k^i\}$,*
- $\mathcal{F} = \{f_1, \ldots, f_m\}$ *is a set of soft constraints, where f_i is a function that defines a cost $\in \mathbb{R} \cup \{\infty\}$ for each combination of values to the variables in it's scope,*
- $\mu : \mathcal{X} \rightarrow \mathcal{A}$ *is a function that assigns the control of each variables to an agent.*

*A solution to the **DCOP** is a complete assignment σ that minimizes a global objective function $F(\sigma)$ that aggregates the individual costs function f_i.*

$$\sigma^* = \underset{\sigma}{\operatorname{argmin}} F(\sigma)$$

The sum is generally used as an aggregation function:

$$\sigma^* = \underset{\sigma}{\operatorname{argmin}} F(\sigma) = \underset{\sigma}{\operatorname{argmin}} \sum_{f_i \in \mathcal{F}} f_i$$

It should be noted that, without loss of generality, the notion of cost can be replaced by the notion of utility $\in \mathbb{R} \cup \{-\infty\}$. In this case, solving a **DCOP** is a maximization problem of the overall sum of utilities.

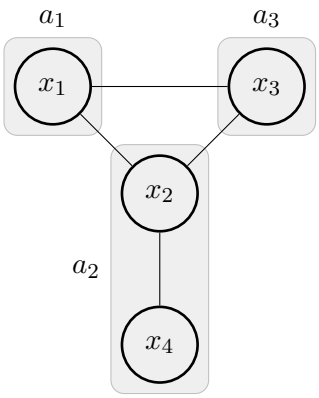
Based on this definition, the following three key characteristics of a **DCOP** justify the affiliation of this framework to **MAS**:

- Each variable is *owned* exclusively by an agent and conversely, an agent *controls* only the variables it owns. This notably implies that an agent can only select a value for, and observe, its own variables.
- An agent is only aware (at least initially) of the constraints whose scope includes a variable it is controlling.
- Agents only know their neighbors, where two agents are considered to be neighbors if there is at least one constraint f_i whose scope contains a variable controlled by each of these agents. Agents interact exclusively with their neighbors, by sending messages to each other (synchronously or asynchronously).

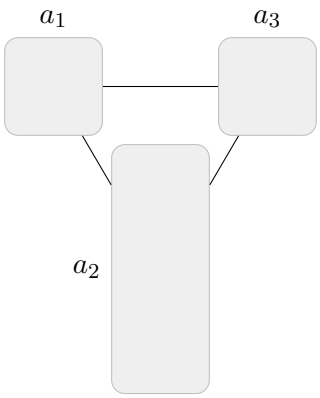
Finally, as for **CSP** and **COP**, a **DCSP** can always be mapped to an equivalent **DCOP**[17] by encoding hard constraints as soft constraints with infinite cost.

Like with classical constraints optimization, finding a solution for a **DCOP** is NP-Hard in the general case, which explains the number of approximate algorithms proposed in the literature.

17. This might however be less efficient, as properties of the hard constraints will not be used



(a) Constraint graph with agents represented as compound nodes



(b) Agents' communication graph

Figure 2.4 – Constraint graph representations for **DCOP**

Several solution methods for **DCOP**, both complete and incomplete, are presented in Section 2.4.

2.3.2.1 Graphical Representation

Like for a **Constraint Reasoning Problem (CRP)**, **DCOP**s and **DCSP**s are also often represented using graphical models. Constraint graphs (see Section 2.3.1.1) are commonly used to represent **DCOP** and are defined like for non-distributed problems, with the addition of *enclosing nodes* (a.k.a. compound nodes) representing the agents and the variables they are responsible for –see Figure 2.4a for a example of such a representation. These nodes form a communication graph among agents, as depicted in Figure 2.4b.

Example 2. *In order to apply the graph coloring problem introduced in Example 1 in a distributed setting, one only has to define agents and a mapping function.*

Let $\mathcal{A} = \{a_1, \ldots, a_7\}$ be a set of agents, responsible for selecting the color of the region-variables, and $\mu : \mathcal{X} \to \mathcal{A}$ a mapping that assigns one variable, in alphabetical order to each agent. This mapping can be visualized on the graphical representation (see Figure 2.5) by placing each variables into an enclosing node that represents the agent responsible for it, forming a compound graph.

When solving the problem, each agent will only interact with its direct neighbor in the graph and no agent has a global view of all the variables and constraints.

Factor graphs are also used for **DCOP**s. As constraints (a.k.a. factors) are explicitly represented in these graphs, they must also be attached to agents, as represented on Figure 2.6.

Depth-First Search Trees (DFS Trees), also known as *pseudo-tree*, are another graphical representation used for studying **DCOP** which is used by many algorithms like ADOPT, DPOP and NCBB.

A **DFS Tree** $T = \langle \mathcal{F}, E_T \rangle$ of a constraint graph G is a rooted tree that is built using a depth first search traversal of the graph. Such traversal yields a spanning tree of the constraint graph, where variables involved in the same constraint must appear in the same branch of the tree. Edges in E_T are called *tree edges* and link parent nodes to children nodes, while edges of G that are not in E_T

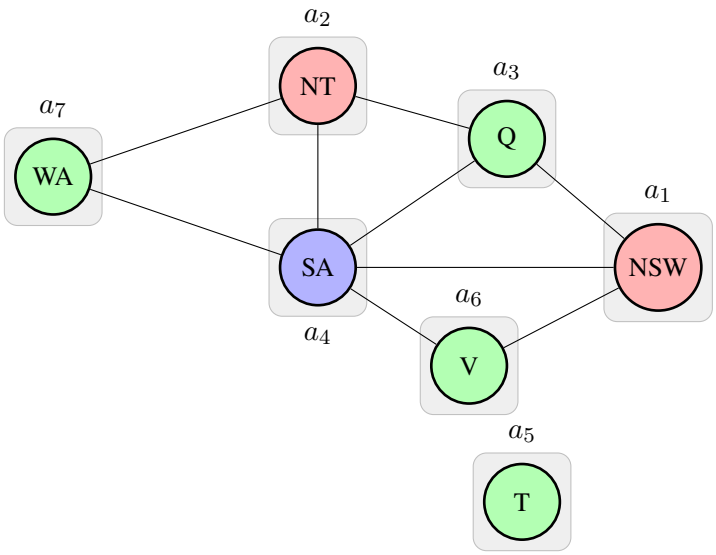
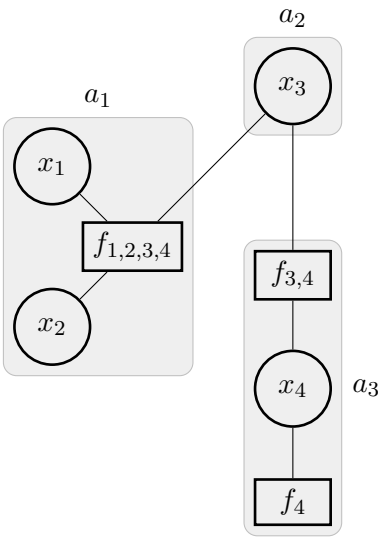
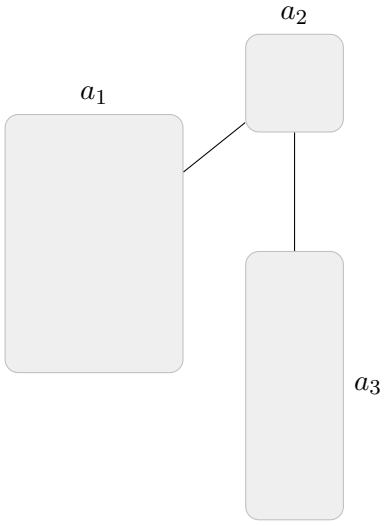


Figure 2.5 – Distributed map-coloring problem on Australia

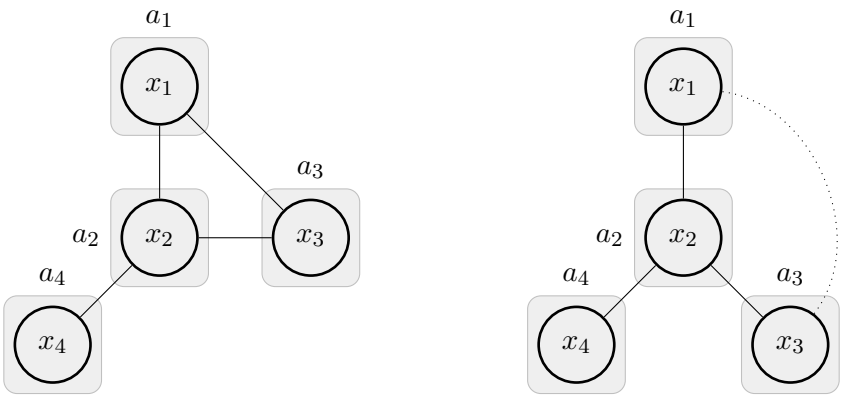


(a) Simple constraint graph with only binary constraints



(b) Agents' Communication Graph

Figure 2.6 – Factor Graph representation for **DCOP**



(a) Constraint graph (b) Depth First Search Tree

Figure 2.7 – The same problem represented with a constraint graph and a **DFS Tree** (backedges are depicted with dotted lines)

are called *backedges* and define as a *pseudo-parent* relationship (resp. *pseudo-children*).

The advantage of this representation, compared to factor graphs and constraint graphs, is that it provides a partial ordering among variables, which is required by many algorithms. Like constraint graphs, **DFS Trees** cannot directly represent non binary constraints, although some algorithms based on this structure have variants to manage such constraints. Many different **DFS Trees** can be generated from the same constraint graph and the efficiency of **DFS Tree**-based algorithms usually depends on the quality of the **DFS Tree**, especially its *depth* (the number of nodes on the longest path) and its *induced width*. Many distributed algorithms have been developed to generate good **DFS Trees**, for example [22, 24].

2.3.2.2 Common Assumptions

Most works in the literature use the three following assumptions, which simplify the algorithmic definitions and can be applied without loss of generality.

Binary Constraints. The first assumption, which is also generally applied for non-distributed constraint reasoning, is that all constraints in the problem are between two variables. As a matter of fact, any constraint network containing constraints of arbitrary arity can be mapped to an equivalent binary constraints network that contains only binary constraints. Two general methods are known (see [29]) for this conversion: the *dual graph method* and the *hidden variable method*.

- When applying the dual graph transformation to the problem, each constraint of the original problem is mapped to a variable whose domain is the set of partial assignments allowed by the constraint. Binary constraints are introduced between every pair of constraints that share a variable, in order to enforce the equality of the value of the original variables.
- The hidden variable methods consist in adding, for each non-binary constraint, one auxiliary (a.k.a hidden) variable whose domain is the Cartesian product of the domains of the variables in the scope of the constraint. A set of binary constraints are also added to ensure that the value in the hidden variable does no contradict the value of the original variables.

Agents-Variable Bijection. The second commonly used assumption is that each agent controls exactly one variable. It is also easy to reformulate a general **DCOP** into one where each agents controls exactly one variable; the two approaches presented for **DCSP** in [156] also apply to **DCOP**. One technique, called *compilation*, is to create for each agent a new variable whose domain is the set of solutions to the local problem defined by the original variables owned by the agent. Another solution, known as *decomposition* is to simply have agent create local *virtual agents*, hosted by the real agents, that correspond to local variables.

Reliable Communication. The last assumption deals with the communication between agents; message delivery is assumed to be reliable: messages sent by agent a_i to a_j are delivered in finite time and messages are received by a_j in the order they were sent.

However, it can be argued that these assumptions do not apply very well when modeling real-world problems. This will be discussed further in Section 4.1.

2.3.2.3 Extensions to the Canonical DCOP Framework

Many extensions have been proposed to the **DCOP** framework, and we briefly introduce here some of the most notable ones.

Dynamic DCOP (Dyn-DCOP)

In the classical **DCOP** framework the problem is fixed and the solving process only needs to run once to obtain a solution. The **Dyn-DCOP** model focus on situation where the problem changes over time: variables' domains and constraints' cost functions may change, agents may leave and enter the system at any time, etc. The idea is to better model real-world problems where the environment is dynamic. A **Dyn-DCOP** is generally defined as a sequence of **DCOP**, with changes between them. Solving the **Dyn-DCOP** means finding a solution for each of the **DCOP** in the sequence, with the goal of solving these problems at least as fast as the environment changes. This extension will be discussed further in Section 5.1.3.

Asymmetric DCOP (A-DCOP)

In **DCOP**, constraints are considered to be symmetric, which means that the cost incurred by a constraint for a given assignment is the same for all agents controlling one variable in the scope of the constraint. In an **A-DCOP**, a constraint may incur a different cost to two agents for the same join assignment. This asymmetry allow modeling problems where agent have different preferences, which they want to keep private. Although it is always possible to transform an **A-DCOP** to a symmetric one (by introduction extra "mirror" variables), several specialized algorithms have been designed to solve these problems more efficiently.

Multi-objective DCOP (MO-DCOP)

MO-DCOP is a framework at the crossroads between **DCOP** and multi-objective optimization. This model has been designed for problems where decisions need to accommodate multiple potentially conflicting objectives. In a **MO-DCOP** constraint's functions are replaced by sets of objective functions and a solution is a complete assignment minimizing the cost vector resulting from these objective functions. Typically there is no single solution that optimize simultaneously all the objectives; the concept of *Pareto optimality* can be applied to the set of solutions to define a *Pareto front*.

Probabilistic DCOP (P-DCOP)

in a **P-DCOP** agents only have a *partial* knowledge of the cost function of the constraints. Agents must at the same time *explore* their environment, in order to discover these cost functions, and *exploit* this knowledge to optimize the global objective.

Quantified DCOP (Q-DCOP)

While the classical **DCOP** framework assumes all agents are acting cooperatively to reach a common objective, in the **Q-DCOP** model, introduced in [82], some agents can be partially cooperative or competitive. Consequently, a **Q-DCOP** does not have one optimal cost (or utility) but defines lower and upper bounds on the cost of a solution, and any solution whose cost is between these bounds is acceptable.

2.4 DCOP Solution Methods

Classical **DCOP** are well studied and a large number of algorithms, many of which have several variants, have been proposed by the research community. In this section we will not be able to give a detailed description of all algorithms, we redirect the reader to [38] for a very exhaustive

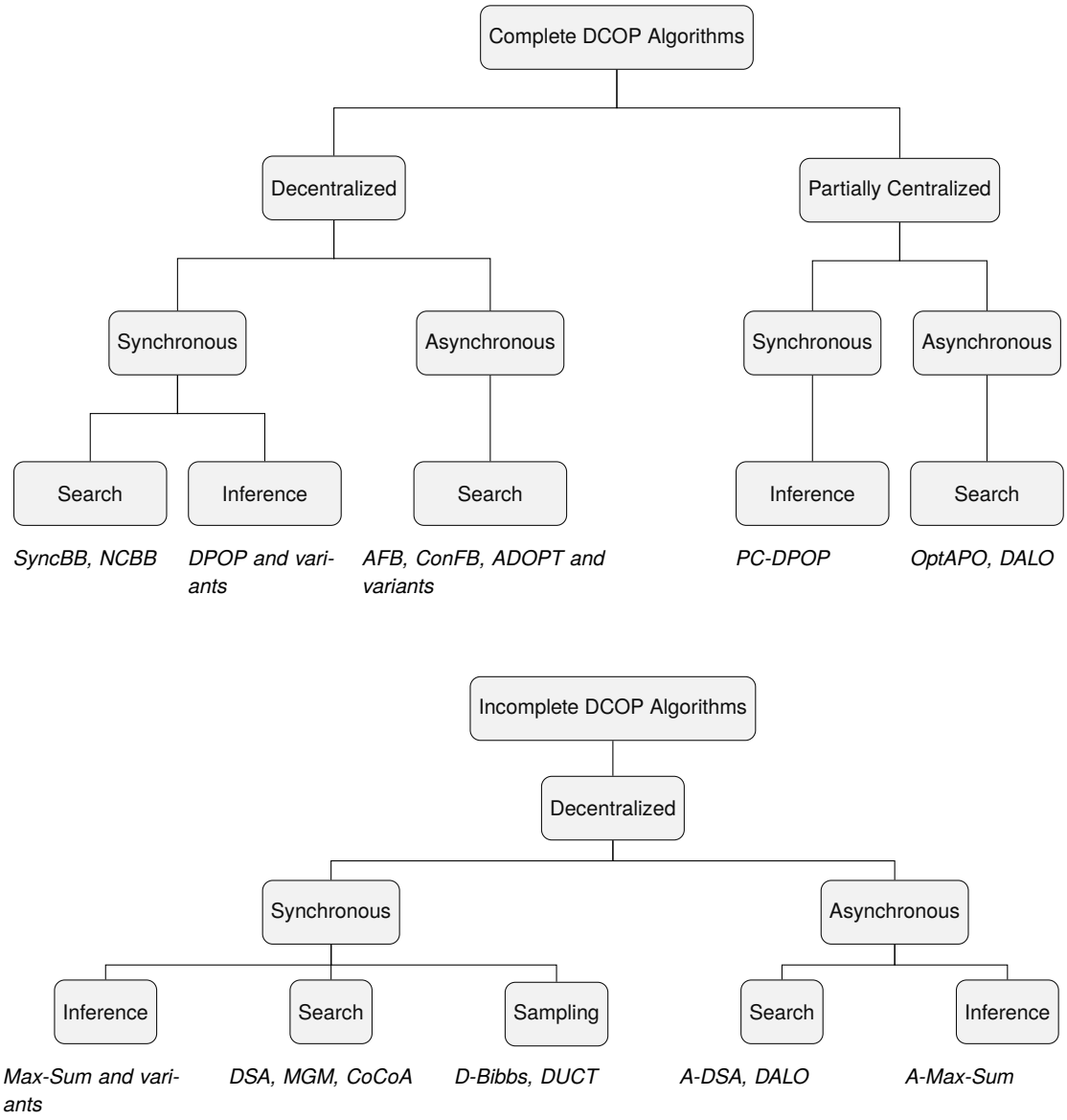


Figure 2.8 – DCOP taxonomy

survey of **DCOP** and to the original papers for each algorithm. We give here a taxonomy of the main DCOP algorithms, illustrated by Figure 2.8. Then we focus on the very algorithms that have been used the most during this study, namely DPOP, Max-Sum, DSA and MGM.

2.4.1 Taxonomy of DCOP Algorithms

DCOP algorithms can be classified according to different criteria, as depicted by Figure 2.8 (inspired by [38]). Here we look at several of these criteria, namely optimality, synchronicity, exploration mechanism and level of distribution.

2.4.1.1 Optimality

The simplest criteria for classifying **DCOP** algorithms is to look at the quality of the results they provide.

Complete algorithms

Some algorithms are *complete*: they are guaranteed to terminate and return a result that is a solution to the DCOP, i.e. a complete assignment that optimizes the objective function. ADOPT [86], DPOP [104], AFB [43], SyncBB [49], NCBB [21], and OptAPO [79] are some well-known complete algorithms.

Approximate algorithms

Some other algorithms, are *incomplete*; they return a result that is only a *near-optimal* assignment. By trading optimality in exchange of lower computational, memory and communication footprint, they are generally able to find a result faster and with less memory and communication load. Max-Sum [36], DSA [158] and MGM [77] for example are incomplete and provide no theoretically proved guarantee bounds, but are very lightweight.

Bounded approximate algorithms

Some approximate algorithms, like Bounded Max-Sum [118], Bounded-ADOPT [86], DUCT [93] and D-GIBBS [90], are able to provide **bounds** on the optimality of their results, meaning that the algorithm can deliver a solution whose quality is within a specified distance of the optimal.

Region optimal approximate algorithms

Some other algorithms provide what is called **region-optimal** results (also called k-optimality [98]): the assignment is not optimal for the whole problem but is optimal for some user-specified regions of the constraint network. This is the case of DALO [63], DSA-k and MGM-k [77].

2.4.1.2 Synchronicity

In the general case of distributed message-passing systems (see [75]), a system is said to be asynchronous if there is no fixed upper bound time limit for message delivery. On the other hand, in a synchronous model transmission times are bounded and the execution of an algorithm is partitioned into rounds: each processor (a.k.a. agent in the case of **DCOP**) can send messages to its neighbors, and computation happens once all the messages have been delivered.

Asynchronous algorithm

In the context of DCOP algorithms we extend these definitions by stating that in an asynchronous algorithm agents make decisions based on their local view of the problem and do not need to wait for the decisions of other agents.

Synchronous algorithm

In contrast, in a **DCOP** synchronous algorithm agents base their decision on the decision of their neighbors and generally follow a particular order, waiting for specific messages before moving to the next step of the algorithm.

As discussed by Peri and Meisels in [101], asynchronous operation tend to allow a better concurrency of decisions but might lead to performing irrelevant computations, as the local knowledge of agent may be outdated. Conversely, synchronous algorithms often result in higher idle time, when agents are waiting for their neighbor's messages before taking a decision.

Additionally, several algorithms that have been designed as synchronous also have an asynchronous variant. This is for example the case with DSA: [40] gives a description of asynchronous-DSA.

2.4.1.3 Exploration Mechanism

The resolution process have been studied in the context of **DCOP** can be classified in three categories: search, inference, and sampling.

Search approaches

The first, and most classical, resolution process is to use search techniques to explore the solution space. The set of (potentially incomplete) assignments is represented as a graph, where each assignment is a node, and the search process moves from one node to another while looking for an assignment that minimizes the constraints. To avoid exploring the whole solution space various techniques have been developed that allow cutting out sub-sets of this space that are guaranteed not to contain an optimal solution. These traditional techniques, developed for centralized search and constraint optimization, have been adapted to work in distributed settings. For example many **DCOP** search algorithms, like **Asynchronous Forward Bounding (AFB)** [43] and **Concurrent Forward Bounding (ConcFB)** [89], are based on the *Depth-First*, *Branch-and-Bound* principles while **Asynchronous Distributed OPTimization (ADOPT)** [86] uses the *Best-First* search scheme.

Local search approaches

As the complexity of the problems (which depends on constraint graph and domains' size) increases, complete search algorithms become prohibitive. A solution to that problem is to let each agent reason only on its local knowledge of neighbor's states and constraints, performing what is called a *local search*. When performing a local search, an algorithm starts with a (potentially random) candidate solution and *iteratively* moves from one solution to another refining *neighbor solution*. The selection of this neighbor solution is only based on local information. Such search processes are generally computationally cheap, but might get trapped in a local optimum, where no neighbor solution is better that the current one, even though the current solution is not optimal. Consequently, algorithms based on this principle are incomplete but scale very well and are able to find near-optimal solutions even for very large problems. DSA (see Section 2.4.2.1), MGM (see Section 2.4.2.2) and CoCoA [73] are examples of such local search **DCOP** algorithms.

Inference approaches

Some algorithms are based on *inference*: they work by propagating messages that summarize the influence of the sending agent (and the preceding sub-graph or subtree) on the rest of the problem. Once enough information on these influence is known, agents can take a decision on the value of the variable they own. These algorithms fall under the framework of the **Generalized Distributive Law (GDL)**, also called *belief propagation*. This category includes both complete, like DPOP (see Section 2.4.2.4), and approximate algorithms, like Max-Sum (see Section 2.4.2.3).

Sampling approaches

A few approximate algorithms, like DUCT [93] and D-Gibbs [90], have been designed by

adapting centralized sampling algorithms (respectively **Upper Confidence bound for Trees (UCT)** and Gibbs) to the distributed setting of **DCOP**. These algorithms sample the search space to approximate a probability distribution function for the **DCOP** solution.

2.4.1.4 Distribution

It might seem strange to consider *distribution* as a criteria for *distributed* optimization algorithms. However, several works have shown that introducing some partial centralization was both an acceptable and efficient approach, as long as the loss of privacy was not an issue in the target application domain.

Fully distributed

Most **DCOP** algorithms fall into this category; has a matter of fact, the **DCOP** framework states that agents may only communicate with their neighbors and only know the constraints they are involved in.

Partially centralized

In partially centralized algorithm, some agents are selected to solve a sub-part of the original problem and advise other agents on interesting value changes. The idea of these approaches is to simultaneously exploit the speed of centralized methods and reduce the communication load, while still preserving some distribution and privacy. **Optimal Asynchronous Partial Overlay (OptAPO)** [79] and **Partial Centralization DPOP (PC-DPOP)** [103] are two examples of partially distributed **DCOP** algorithms.

2.4.1.5 Solution Availability

Another way to differentiate **DCOP** solution methods is to look at the way they make their solution available, potentially providing intermediate solutions.

Most complete algorithms only provide their solution at the end of the solving process. While their solution is complete, for complex problems the runtime might be prohibitive. Additionally, this family of algorithms is usually impractical in dynamic settings, as the problem might have changed before any solution is available.

Some other solution methods, called *iterative algorithms*, can provide intermediate results during their execution. With this mode of operation, intermediate solutions are of course approximate, even if some algorithms, given enough time, would reach the optimal solution. Some of these iterative algorithm, like MGM, are *anytime* (i.e. monotonous and iterative): they guarantee that the solutions they provide will always be of better or equal quality over time. Some other algorithms, like DSA, do not provide such guarantee, even though it can be shown experimentally that the quality of the results increases *on average* over time.

It should be noted that some algorithms, like for example inference algorithms, do not produce a usable solution directly but require some decoding.

2.4.2 Some DCOP Algorithms

As it can been seen from previous sections, far too many **DCOP** algorithms have been proposed to describe them all precisely here. Instead, we choose to concentrate on the four algorithms that have been used the most in this study, namely **DSA**, **MGM**, **MaxSum** and **DPOP**.

The first three algorithms have been selected because they are lightweight approximate algorithms that are particularly suited for constrained connected objects. DPOP, on the other hand, is a complete algorithm with a significant computational cost, but is very useful in our case as its optimality gives us a reference to evaluate the quality of the results obtained when using other approximate algorithms.

2.4.2.1 DSA

More than a single algorithm, **Distributed Stochastic Algorithm (DSA)** is a family of incomplete, local search, very lightweight algorithms based a rather simple idea: agents start with a random value from their domain and regularly evaluate if the quality of their own partial assignment, defined as the total values of those constraints in which it is involved, could be improved by selecting a new value [157]. This evaluation is based on the knowledge of the values currently selected by their neighbors. If this quality can be improved, the agent decides randomly, with an *activation probability* p, to select the corresponding value and send its updated state to its neighbors.

This search process is *local* as agents base their decision only on their knowledge of the values of their direct neighbors. Of course such local search algorithm can become trapped in a local minima (even if **DSA** stochasticity sometime helps it escaping such local minima) and does not guarantee to find the optimal solution. Although not strictly *anytime*, **DSA** is an *iterative* algorithm and can be used to obtain a complete assignment at any time, in real time, with a solution quality improving, on average, over time. However, in the general case **DSA** provides no guarantee of monotony: as there is no coordination in the decision process and an agent's local knowledge may be outdated, two agents may simultaneously take contradictory decisions, resulting in a decrease in the overall result's quality.

Five variations –namely **DSA**-A, **DSA**-B, **DSA**-C, **DSA**-D and **DSA**-E– of this basic principle have been studied [157], depending on the strategy used for values change. An agent may select a new value more or less aggressively, when its state quality can be improved, strictly or not, and when where are still conflicts even if the quality cannot be improved. These variants exhibit various degree of parallelism and solution space exploration. **DSA**-B is considered to be the most efficient approach in the general case.

The value used for activation probability p has also been shown in [157] to have a huge influence in **DSA**'s efficiency and quality and exhibits phase transition property. When the right variant and activation probability have been selected for a given problem class, **DSA** provides very good quality results, with minimal network and computational load, which makes it highly scalable.

It should be noted that **DSA** is able to work with n-ary constraints without any modification.

Depending on the time at which the agent's decision process take place, two modes of execution are possible:

Synchronous

When not explicitly mentioned otherwise, **DSA** refers to the synchronous version of the algorithm: all agents proceed in synchronized rounds. In each round, each agent send its current value to its neighbors and, once it has received the value from all its neighbors, takes a decision about its own value change. The process is repeated continuously until some termination condition is reached, usually a predetermined number of rounds.

Asynchronous

DSA can also be implemented in a completely asynchronous fashion (see [40]): each agent sends its value and take its decision, based on its current knowledge, at a random periodicity. This approach has been shown experimentally to provide good results, as long as the rate of change is low enough to allow propagation of information in spite of the communication latency.

A coordinated variation of **DSA**, SCA-k, has also been proposed in [77].

2.4.2.2 MGM

Maximum Gain Message (MGM) is a modification of **Distributed Breakout Algorithm (DBA)** that focuses on gain message passing [77]. Like **DSA**, **MGM** algorithm is an incomplete local search algorithm that can handle n-ary constraints

MGM is a synchronous algorithm: at each round, agents compute the maximum change in quality, named *gain*, they could achieve by selecting a new value. and send this gain to their neighbors. An agent is then allowed to change its value only if its gain is larger than the gain received from all its neighbors. This mechanism ensure that two variables involved in the same constraint will never change their value in the same round. This process repeats until a termination condition is met.

While it provides no bounds on the solution quality, **MGM** is able to guarantee monotony; eliminating the stochastic aspect of **DSA** ensures that the solution quality only improves over time. Monotony is a very interesting quality in many application domains, however, this quality is guaranteed at the expense of an higher tendency to become trapped in a local minima.

To mitigate this issue, [77] proposes a coordinated version of **MGM** (usually MGM-2, but it can be extended to MGM-k), where k agents can coordinate a simultaneous (i.e. in the same round) change of values. This allows avoiding some local minima while preserving the monotony of the algorithm.

Although it should be possible to implement an asynchronous version of **MGM**, to the best of our knowledge no such variant has been proposed yet.

2.4.2.3 MaxSum

MaxSum is a inference-based incomplete algorithm [36]. It is a derivative of the *max-product* message passing algorithm in the logarithmic space. **MaxSum** is complete on acyclic constraint graphs, but approximate on cyclic graphs.

It operates on a factor graph (FG) (see Section 2.3.1.1) and messages flow on the edges, from factors to variables, and vice versa:

- A message from factor f_m to variable x_n is a vector $R_{m\to n} = [r^1_{m\to n} \dots r^k_{m\to n}]^\intercal$, with $k = |\mathcal{D}_n|$ where:

$$r^i_{m\to n} = f_m(v_i) + \max_{X\setminus x_n} q^i_{n\to m}$$

- Similarly, a message from variable x_n to factor f_m is also a vector $Q_{n\to m}$ defined as follows, where α_{nm} is a scalar used to normalize messages (usually selected such that $\sum_{0<i<|\mathcal{D}_k|} q^i_{n\to m} = 0$) and avoid their values to grow indefinitely, as messages propagate in loops in a cyclic graph.

$$q^i_{n\to m} = \alpha_{nm} + \sum_{\{f_p | x_n \in S_p\}\setminus f_m} r^i_{m\to n}$$

When a factor or a variable computes twice the same message for the same recipient, it stops propagation and the algorithm converges if all message propagation has stopped. However, while **MaxSum** is guaranteed to converge on acyclic constraint graphs, it may not converge at all on cyclic graphs. Consequently termination is usually implemented by both observing convergence and by force-stopping propagation after a predetermined number of rounds.

By simply summing its $R_{m\to n}$ messages an agent can assess **at any time** an approximation of the marginal function of the variable x_n and select the value that maximizes the social welfare in the system by finding the argmax of this marginal function. This also means that Max-Sum can be used to get a continuously updated solution, without waiting for termination, even in dynamic problems.

Empirical evaluations show that it can compute very good quality solutions with acceptable computation load compared to representative complete algorithms.

Several approaches have been studied to increase the quality of the solution produced by **MaxSum**, like introducing *noise* in cost for easier tie breaking, applying *damping* [23] on the messages, to facilitate convergence in the presence of cycles, or decimating variables by assigning values and removing variables depending on their marginal values, at runtime [20].

Additionally, many derivative of **MaxSum** have also been proposed, like **MaxSum** ADVP [160] and Bounded-**MaxSum** [118], to handle cycles by processing inference over trees or directed acyclic graphs, instead of the initial cyclic factor graph.

Finally, while most works describe and classify **MaxSum** as a synchronous algorithm, it should be noted that it can also be implemented asynchronously, as stated in [36], with agents emitting updated messages whenever they receive an update from on of their neighbors. We denote this variant **Asynchronous MaxSum Algorithm (A-MaxSum)**. However, this requires some measure to avoid an excess of messages, which are not well studied yet to the best of our knowledge. **MaxSum** is also well suited to dynamic settings, as the agents can maintain an up-to-date estimate of the current state's utility by continuously emitting update messages.

2.4.2.4 DPOP

The **Distributed Pseudo-tree Optimization Procedure (DPOP)** is an optimal, inference-based, **DCOP** algorithm implementing a dynamic programming procedure in a distributed way [104] and

can be seen as an extension of the general bucket elimination scheme [28].

DPOP mainly runs three phases:

1. It builds a **DFS Tree** that overlays the constraint network (see Section 2.3.2.1). This pseudo-tree, made of parent links and pseudo-parent links (when loops appear in the constraint graph) is used by agents owning variables to interact during the next phases. This phase can be implemented using a simple token-passing protocol.
2. Once the **DFS Tree** is built, cost functions are sent from the leafs up to the root. Agents assess the *cost messages* they send to their parent by joining all the messages received from their children. A cost message sent by an agent is a multi-dimensional hash map (or hypercube) associating a cost to every possible value of its parent and pseudo-parents. Computational complexity of each agent in DPOP is exponential on the number of pseudo-parents, fundamentally due to the assessment of the cost messages, which represents an obstacle when coping with cyclic graphs.
3. Once the root has received the cost messages from its children, it assesses the aggregated costs of the whole problem and then it decides the best assignment for its variable. Finally, it broadcasts this assignment in a *value message* to its children, who assess their best assignments and send them down to the leafs.

Notice that when running on a tree-shaped graph DPOP and Max-Sum, which are both based on *belief propagation*, are strictly equivalent.

After the execution of the algorithm, each agent knows the optimal values for the variables in its scope. DPOP returns an optimal assignment, with only a linear number of messages. Many DPOP extensions and other exact algorithms work in a similar way [144].

Many variants of DPOP have been proposed to trade runtimes for smaller memory requirements [102, 103], trade solution optimality for smaller runtimes [105], trade runtime for increased privacy [45], trade privacy for smaller runtimes [103], propagate hard constraints for smaller runtimes [68], or enforce branch consistency for smaller runtimes [37].

2.4.3 Evaluating the Performance of DCOP Algorithms

Evaluating the performance of a **DCOP** algorithm is a complex subject. This evaluation is generally motivated by the need to estimate the time between the starting time of the process and the time we get an acceptable solution. However, simply comparing this time across algorithms is not a good method for comparing their respective performance, as in this case these measures are mostly representative of the efficiency of the implementations [18] of these algorithms and not of the algorithm themselves.

To avoid this issue, the *constraint reasoning* community traditionally compares algorithms by counting the number of *constraints check* (CC) needed to reach an acceptable result. This measure is machine and implementation independent as it counts an operation that would be performed by any implementation of the same algorithm and is generally representative of the overall complexity

18. This is even more problematic as there is currently no software library that implements most of the **DCOP** algorithms, which means that one would compare different implementations made with different languages and probably developed with different assumptions and objectives.

of the problem.

However, counting CCs is less meaningful in *distributed* constraint reasoning; it does not allow to estimate the resolution time as, depending on the algorithms, various degree of parallelism may be observed in these constraints checks. Additionally counting CCs also does not take into account delays caused by messages: agents need to send and receive messages and may need to wait for messages to perform their own computations.

One proposed metrics is the number of *Non Concurrent Constraint Check* (NCCC), which represents the length of the longest computation chain that cannot be executed concurrently [84] but does not take into account communication delays, which in a real deployment may be many orders of magnitude higher than the computational effort performed in a single step [26]. Other alternatives have tried to combine latency with NCCCs [159], however the evaluation of **DCOP** algorithms performance is generally still considered to be an open question.

2.5 Application of Constraint Reasoning for Ambient Intelligence

Several researchers have used constraint reasoning approaches, both centralized and distributed, for tackling **Ambient Intelligence (AmI)** and **SHE** problems. Most works concentrate on the use of constraints for planning and for considering user preferences. We provide in this section a short review of some related works in this area. Section 2.5.1 focus on those based on traditional constraint reasoning; works based on distributed constraint reasoning are discussed in 2.5.2.

2.5.1 Constraint Reasoning in Ambient Intelligence

In [30], Degeler et al. use dynamic constraint reasoning for smart environment management, citing the flexibility and adjustability of constraint reasoning as the reasons for this choice. The desired behavior of the home is specified using logical rules, which combine the context information about the environment with the expected actions of connected devices. The problem is then encoded as a **Dynamic Constraints Satisfaction Problem (DynCSP)**, where actuators are represented as *controllable variables*, in order to take into account the changes in the environment context. Solving this **DynCSP** continuously yields updated values for these variables, which map to a state and/or action of the corresponding actuator(s). By building a dynamic dependency graph, it is possible to identify the sub-problems that are really impacted by a change in the environment and only invoke re-optimization tasks for the smallest subsets of the variables which are actually affected, effectively reusing parts of the earlier solutions. This approach allows the system to scale up, performing real-time even with hundreds of variables.

Kaldeli et al. argue in [60] that intelligent behavior in a **SHE** requires complex functionalities that involve several dynamically selected services provided by independent devices. To achieve this kind of behavior, they propose to combine a **SOA** design with automatic and dynamic service composition. The service composition is implemented using a domain-independent **CSP**-based planner. The home domain is represented as a dynamic constraints network, constraints being added and removed depending on the current state of the environment and the set of available services and devices. The planner reasons upon this model and generates a sequence of actions that must be performed. The authors justify the choice of a CSP planner by its performance,

its adaptability and its ability to perform complex reasoning about contextual information. They also advocate that constraint based systems are well suited for user-centric environments as they allow declarative high-level goal specification, instead of requiring a description of how the request behavior should be achieved.

In [135], Song et al. design a self-adaptive system that takes user preferences into account and apply it to a **SHE** scenario. The set of adaptation policies (i.e. the changes needed to meet the user's goals) is modeled as a **CSP**, which allows detecting conflicting goals and finding the best configuration that satisfies as many goals as possible. When the user change their preferences, the weight of the existing constraints are automatically tuned accordingly, and new constraints are generated when required.

Parra et al. use a **CSP** to model a dynamic features selection in a software product line and optimize this configuration process [96]. This work is applied to a **SHE** scenario where an adaptive application must self-adapt to the type of devices and connectivity options currently available in the house.

2.5.2 Distributed Constraint Reasoning for Ambient Intelligence

As discussed previously, a large body of works can be found on the use of **MAS** for **AmI** and **SHE**. Similarly, some research has been done on applying the **DCOP** framework to settings that can be considered to be part of **IoT** and Ubiquitous computing: sensor networks, radio frequency allocation, traffic light coordination, etc. However, to the best of our knowledge, few past works have focused on the use of the **DCOP** framework for **AmI** and **SHE** settings. We discuss here two of these works.

In [99], Pecora et al. argue that the integration of complex services for **AmI** is a problem that requires intelligent reasoning and should be solved with AI problem solving. They advocate that service coordination can be mapped to **Multi-Agent Coordination (MAC)**, which can be tackled using the **DCOP** framework. In their model, each application in the home offers one or several services and is represented by an agent. These applications are a combination of sensors, actuators and symbolic reasoning processes. Variables, both input and output, are used to represent the external interface of these services and constraints are used to model the functional relations among them. The resulting DCOP is continuously reasoned upon by the agents and yields a solution where output variables map to the desired behavior of the home. Their implementation is based on ADOPT-N [100], an extension of the original ADOPT algorithm designed to address problems with n-ary constraints, whose extension may not be known *a priori* (this is called *constraint posting*).

Fioretto et al. propose in [39] to use a **DCOP** approach for demand-side management in electric smart grid, based on the Smart Home Device Scheduling (SHDS) problem. They consider multiple smart homes, each equipped with smart devices and their goal is to find an activation schedule for all devices that achieve used-defined preferences (light, temperature, etc.) while at the same time optimizing the aggregated cost of energy consumed. As the cost of energy is a function of time, this method help avoiding peak in energy consumption, especially at times when the price is high. This problem is modeled as a distributed scheduling problem, which is mapped to a **DCOP** that includes both soft constraints, for user preferences and energy consumption, and hard constraints,

for temporal goals. This use of distributed optimization provide coordinated schedules across several homes, which is necessary for efficient demand-side management, while also preserving the privacy of the users of the smart homes. Their implementation uses SH-MGM, a custom MGM-based algorithm.

As a complement to the SHDS problem, Kluegel et al. propose in [64] a set of physical models for smart home devices and a data set of problem instances that can be used to benchmark solution methods.

2.6 Summary

In this chapter we have expounded the **Ambient Intelligence** paradigm as an application of **Internet of Things** to smart home settings. We have also introduced **Multi-Agent Systems** and more specifically one of its approaches: **Distributed Constraint Optimization Problem**. A brief overview of **Dynamic Constraint Reasoning** has been presented, including majors variants and solution methods.

We have seen that **MAS** have been widely recognized has an efficient paradigm for coordinating the actions of devices and services in **SHE** and **AmI** settings. Moreover, **DCOP** have been used in many works to implement coordination in distributed systems, including many IoT-like systems like sensor networks, smart-grid, etc. However, to the best of our knowledge, the use of DCOP for **AmI** had received few attention for researchers. In the next chapters, we focus the challenges that need to be worked on, in order to enable a better use of **DCOP**-based approaches in in **AmI**, namely the issues of distribution, dynamic and resilience.

3 A Model for Coordination in Smart Environments

As discussed in the previous chapter, we intend to use a **DCOP**-based approach to install distributed, autonomous and spontaneous coordination between devices in **AmI** settings. A first required step in that direction is to devise a model of such environments that can be mapped to an optimization problem. This chapter introduces the **Smart Environment Configuration Problem (SECP)** and expounds how it can be mapped to a **DCOP** in order to be solved by the devices available in the target environment.

3.1 The Smart Environment Configuration Problem

In this section, we start from a sample Smart Home scenario in order to illustrate the coordinated behaviours our model should implement in **AmI** systems and introduce notations required to map these behaviours to an optimization problem.

3.1.1 Sample Ambient Intelligence Scenario

We consider the following **AmI** scenario. Our system is a Smart Home, made of many connected devices, most of which are already commonly available today: various light bulbs and lamps, roller shutters, motorized curtains, TV sets, luminosity sensors (potentially inside and outside the house) and presence detectors, etc. The overall objective of our system is to maintain a luminosity level in the rooms of the house that satisfies the inhabitants, which are considered to be the *users* of the system.

These users can express their wishes by configuring simple behaviors (commonly known as *scenes* in such systems) using an application on a user interface device, like a tablet for example. These scenes can use the values of sensors or the states of some devices as triggers for setting a specific luminosity goal in a given area. For example, one could configure the system such that a luminosity level of 60 [1] is requested in the living room whenever somebody is in this room. Such a configuration can be expressed by a rule as represented in Example 3, although the users would of course not write it in this form, but more probably use some kind of graphical representation to express it.

1. We use abstract units for the luminosity level.

Example 3 (Scene specification)**.** *The rule (3.1) defines a scene where the light level of the living room should be set at* 60 *whenever someone is present in the room:*

$$\begin{array}{lllll} \texttt{IF} & \texttt{presence\_living\_room} & = & \texttt{1} \\ \texttt{THEN} & \texttt{light\_level\_living\_room} & \leftarrow & \texttt{60} \end{array} \tag{3.1}$$

Rule (3.2) refines rule (3.1) by triggering only when the light level is less than 60*:*

$$\begin{array}{ll} \texttt{IF} & \texttt{presence\_living\_room} = 1 \\ \texttt{AND} & \texttt{light\_sensor\_living\_room} < 60 \\ \texttt{THEN} & \texttt{light\_level\_living\_room} \leftarrow 60 \end{array} \tag{3.2}$$

Rule (3.3) refines rule (3.1) by triggering only when the light level is less than 60 *and closing the shutter of the living room as an additional action:*

$$\begin{array}{lllll} \texttt{IF} & \texttt{presence\_living\_room} & = & \texttt{1} \\ \texttt{AND} & \texttt{light\_sensor\_living\_room} & < & \texttt{60} \\ \texttt{THEN} & \texttt{light\_level\_living\_room} & \leftarrow & \texttt{60} \\ \texttt{AND} & \texttt{shutter\_living\_room} & \leftarrow & \texttt{0} \end{array} \tag{3.3}$$

One important characteristic of such rules is that they do not need to contain the list of actions required, but only the objective requested by the user. More specifically, the user does not need here to specify what lights must be switched on or off, nor if the shutters should be opened. This means that our **AmI** system will need to figure out the best actions that would lead to meet the requested objectives. Of course, if one specific action is required by the user, it can also be used as an objective, as demonstrated in rule 3.3 above.

As these actions are not fixed in advance, it also means that the system uses whatever devices available when the scene is triggered: lights bulb might be added or removed, they will be automatically integrated into the solution if needed. Additionally, if a device like a motorized curtain or roller shutter becomes faulty the system automatically adapts the actions of other light emitting devices to take this issue into account; it could even switch the TV on just to get some emergency lighting, if not other source of light is available.

Finally, we want our system to select the most energy-saving configuration for a given scene.

Note that we concentrate here on devices that can influence on the luminosity of the environment, but this approach could be used for many other environmental settings and more generally for almost any behavior in a **AmI** environment.

Each of the connected smart devices in the **AmI** environment acts as a *sensor* or an *actuator*. Of course some devices may exhibit both behaviors. Each device is defined by:

- A unique identifier. As our devices are connected, their MAC address could be used for that purpose.
- Its location in the environment. This would typically be the rooms or areas (some big rooms might be subdivided into several areas) of the home.
- A list of capabilities, like emitting light, producing heat, playing music or videos, etc.

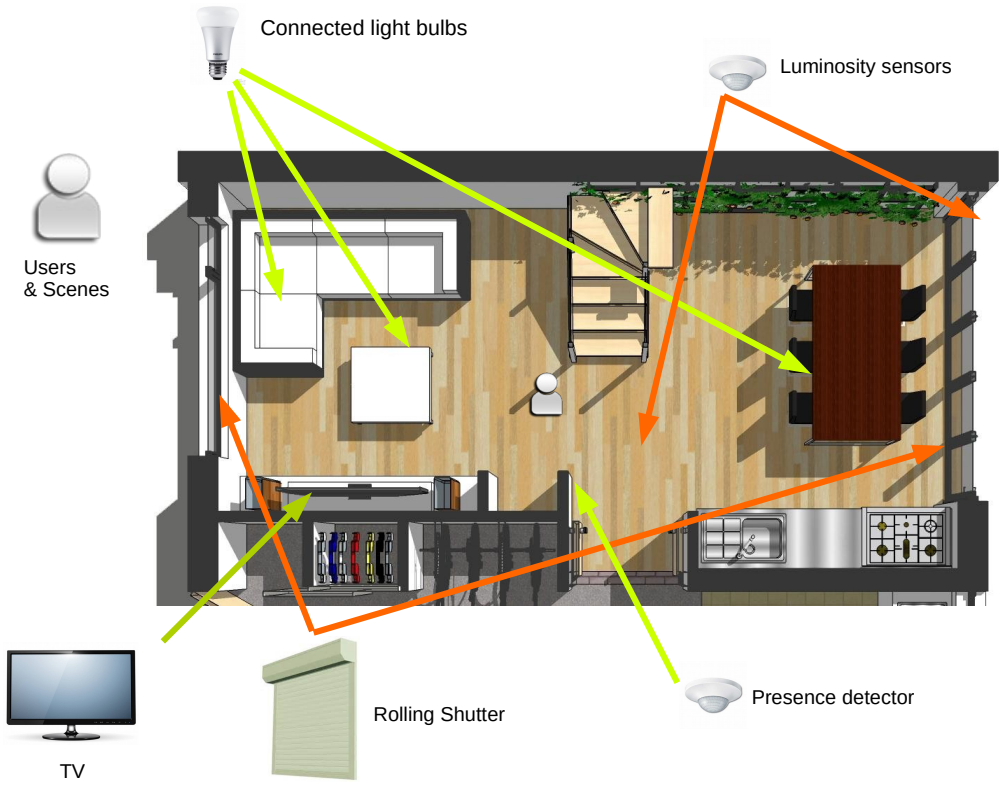


Figure 3.1 – Example of an **AmI** house system with its devices

- A list of actions, if the device is an actuator. For a light bulb, this is for instance the list of light emitting levels achievable (set the bulb at $0, 10 \ldots 100\%$).
- A consumption law associating an energy cost to each action.

Figure 3.1 depicts a simplified smart home equipped with such connected devices.

3.1.2 Problem Definition

Given the scenario described in the previous section, we want our **AmI** system to reach the users' objectives without being steered from a central point, in order to avoid the centralization pitfalls described in Section 2.1.3.1. More precisely, we want the *devices* of this system to self-organize and cooperate autonomously to reach the user-defined objectives, avoiding any dedicated device whose purpose would be to gather inputs from sensors and decide the sequence of actions required to fulfill these objectives.

Devices' capabilities and locations can be used to match candidate devices with a user-defined objective. For example when the rule 3.2 requests a given luminosity level in the living room, one can easily, based on these two elements, identify all devices that can influence the light level in that room.

In the rest of this document we consider that a discovery mechanism is available and that the system knows at any given time the list of available devices with their aforementioned characteristics. Such mechanisms are a common requirement in dynamic open environments and solutions, although far

from perfect, are already available in current systems. For example, mDNS [58] and DNS-SD [57] serve as the technical foundation of discovery functions in current IP-based **SHE** systems.

As a consequence, we concentrate here on the coordination mechanism. We consider each device in the system as an *agent* in a **MAS** and will devise ways of implementing coordination among these agents in a dynamic open system.

3.1.3 Notations for SECP

Our **AmI** scenario can be seen as an optimization problem with values to assign to actuators, whilst maximizing the adequacy to user-defined scenes and minimizing the overall energy consumption.

The notations defined in this section will be used in the remainder of this document. For easier reading, a full reference table of all notations used in this document is given in Appendix A.

3.1.3.1 Actuators

Let $\mathfrak{A}$ be the set of available actuators. We note $\mathcal{X}(\mathfrak{A})$ the set of variables x_i stating the values of actuators $i \in \mathfrak{A}$. We use $\mathbf{x}_i$ to refer to a possible state of $x_i \in \mathcal{X}(\mathfrak{A})$ (i.e. the value assigned to the variable x_i), that is $\mathbf{x}_i \in \mathcal{D}_{x_i}$ where $\mathcal{D}_{x_i}$ is the domain of x_i and contains values mapping to the actions available on the actuator i. For a light bulb for example, this is the list of light levels that can be emitted by the bulb (we consider our light bulb to have a discrete finite list of possible configurations).

Each actuator i has a cost to be activated, noted $e_i : \mathcal{D}_{x_i} \to \mathbb{R}^+$. This cost can be directly derived from the consumption law of each device (e.g. mapping monetary or energy to each action). We note $\mathcal{F}(\mathfrak{A}) = \{e_i | i \in \mathfrak{A}\}$ the set of costs for all actuators. Among the possible values, every actuator i has a possible "switched off" state value, noted $\mathbf{0} \in \mathcal{D}_{x_i}$, with an associated cost (most probably 0).

3.1.3.2 Sensors

Similarly, we note $\mathfrak{S}$ the set of available sensors, and $\mathcal{X}(\mathfrak{S})$ the set of variables s_l encapsulating their states. Each variable s_l take its value in a domain $\mathcal{D}_{s_l}$. We note $\mathbf{s}_l \in \mathcal{D}_{s_l}$ the current state of sensor $\ell \in \mathfrak{S}$. Sensor values reflect the state of the environment and are not directly controllable by the system: therefore they are *read-only* values and do not have a cost function.

3.1.3.3 Environment State

In order for the user to set their objectives (e.g. requesting a given light level in a given room), we also need to model the state of the environment. We note Φ the set of states of the environment, and $\mathcal{X}(\Phi)$ the state of variables y_j encapsulating these states. Each variable y_j takes it value in a domain $\mathcal{D}_{y_j}$ and we note $\mathbf{y}_j \in \mathcal{D}_{y_j}$ the current value of y_j. Like sensors, the environment's state is of course not directly controllable by the system.

3.1.3.4 Scenes

Let $\mathfrak{R}$ be the set of user-defined scene rules. Each scene k is specified as a condition-action rule expressed using the set of available devices $\mathfrak{A} \bigcup \mathfrak{S}$ (actuators and sensors).

The **action** part of scenes defines objectives by setting *target values* to *scene action variables*. These scene action variables can represent either some actuators or the state of the environment:

(1) When the rule requests some direct action on an actuator, the scene action variable is the variable representing the state $x_i \in \mathcal{X}(\mathfrak{A})$ of that actuator. This is for example the case in the rule 3.3 of example 3, which explicitly requires to close the shutter of the living room.

We note $\mathbf{x}_i^k$ the target value defined by the user for the scene action variable x_i in the rule k, with $\mathbf{x}_i^k \in \mathcal{D}_{x_i}$ for all i and k.

(2) When the rule set a target state for the environment the scene action variable is the variable $y_j \in \mathcal{X}(\Phi)$ representing that state of the environment. Of course, there must be some actuators that can act on this state for the rule to have any effect. For example, the three rules in example 3, set a target light level in the living room, which can be acted on by the light bulb actuators located in this room. This approach allows setting goals on more abstract concepts, without hard-coding any specific action nor actuator on the rules; these kind scene action variables actually require some kind of cooperation.

We note $\mathbf{y}_j^k$ the target value defined by the user for the scene action variable y_j in the rule k, with $\mathbf{y}_j^k \in \mathcal{D}_{y_j}$ for all j and k. Note that a scene action variable can be used in several rules, but that a rule can only specify a unique target value for the scene action variable.

The **condition** part of a scene is specified as a conjunction of boolean expressions using state of actuators, $x_i,\ i \in \mathfrak{A}$, or state of sensors, $s_l,\ l \in \mathfrak{S}$ and binary predicates (e.g. $>, <, =$). A scene rule can be either *active* or *inactive* depending on the state of devices appearing in the condition part of the rule.

3.1.4 Modeling Physical Constraints

In order for the system to be able to select the right actions to achieve the goals set by the rules, we must be able to reason upon the link between the actuators' actions and the state of the environment, which means we need a model of the physical interactions happening in the real world. For this purpose, we define functions that we call *physical models*, noted ϕ_k, where $S_{\phi_k} \subseteq \mathcal{X}(\mathfrak{A})$ is the scope of the model, i.e. the set of actuator variables influencing one particular aspect of the environment, and $\mathcal{D}_{\phi_k}$ is the domain of the variable representing the corresponding state of the environment.

$$\phi_k : \prod_{\varsigma \in S_{\phi_k}} \mathcal{D}_\varsigma \rightarrow \mathcal{D}_{\phi_k} \tag{3.4}$$

For example, in the rules of example 3, the scope of the physical model would be the set of light emitting actuators that are located in, and can influence the light level in, the living room.

Example 4 (Physical model)**.** *We can consider that the level of light y_1 in a room depends on the total power of "light-emitting" devices located installed in the room, i.e. bulbs x_1 and x_2, and a*

TV set x_3:

$$y_1 = \phi_1(x_1, x_2, x_3) = 30x_1 + 30x_2 + 10x_3$$

Weights assigned to each x_i are related to the luminous efficacy of each device [136].

In a more general form, a physical dependency model links a set of actuators –generally with the same given capability and in a same given location– to a physical value that can be measured by some sensor.

Let $\overline{\phi_j} = |S_{\phi_j}|$ the arity of ϕ_j, and $\mathcal{F}(\Phi) = \{\phi_j\}$ be the set of all physical models between actuators and rule-defined values.

Of course, the co-domain of ϕ_k depends on the state of the environment considered by this physical model (luminosity, temperature, humidity, etc.). As rules express target for the state of the environment using variables $y_j \in \mathcal{X}(\Phi)$, we need one physical model ϕ_j for each y_j used in the rules, with $\mathcal{D}_{\phi_k} = \mathcal{D}_{y_j}$ and S_{ϕ_k} is the set of actuators influencing y_j. Note that a physical model function output value is considered here to be an *estimation* (or *prediction*) of the value of some y_j, based on the state of the actuators. The quality of the resulting system configuration depends tightly on the quality of this estimation, and thus on how we define these physical model functions.

There are several options to assess the exact functions to be used for these physical models. While this point is not the focus of this work we list here a few approaches that could be used:

- As these functions depend on real world interactions, the physical laws that govern them is generally well known and can easily be found in the corresponding literature. This means that the structure of the function can be fixed in advance, depending on the kind of environment aspect, and we would only need to fill in some parameters like weights.
- These weights could be discovered during a dedicated calibration phase, where the actuators scan their respective action space while we monitor the resulting state of the environment using sensors. With this approach, most sensors are only needed during calibration. At runtime, only sensor used in the rule's condition are necessary. However, when the set of available actuators changes (addition or removal) a new calibration phase might be needed to devise an optimal model.
- Given enough sensors are available, an online machine learning approach could also be used, meaning that the definition of these physical models could improve over time. This solution also has the advantage that the models can dynamically adapt to changes in the environment. For example, the physical model of the luminosity in a room depends on the state of light-emitting actuators but may also depend on the season, when exterior light and sunset times vary.
- More generally, machine learning-based approaches could be used to learn a full model of a physical model from scratch, or to simply learn appropriate weights for a model whose structure is already known from physics [44, 76].
- **AMAS** [132] could also be used to deal with the dynamic and non-linearity exposed by these models, as demonstrated in [13] where **AMAS** are used to perform the automatic calibration of an engine control unit.

In the remainder of this document, we consider that the functions of these physical models are

known and can be used directly. For simplicity, we also consider here these functions to be static, even though we will demonstrate in Section 5.1.3 that using appropriate solution methods dynamic functions could be dealt with.

3.1.5 Formulation as an Optimization Problem

Now that we have a definition of the impact of the actuators' action on the environment, we are able to assess if the objective of a given rule is met. For this purpose, we define for each scene an utility function, noted r_k, with $S_{r_k} \subseteq \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi) \cup \mathcal{X}(\mathfrak{S})$ being the scope of the rule, made of the actuators, sensors and scene action variables used in the rule:

$$r_k : \prod_{v \in S_{r_k}} \mathcal{D}_v \to \mathbb{R}$$

The more the states of the scene action variables (from $\mathcal{X}(\mathfrak{A})$ and $\mathcal{X}(\Phi)$) are close to the user's target values for this scene, the higher the utility. Moreover, if the conditions to activate the rule (from $\mathcal{X}(\mathfrak{A})$ and $\mathcal{X}(\mathfrak{S})$) are not met, the utility should be neutral, i.e. equals to 0. We can therefore consider r_k's to be functions of the distance between the states of the scene action variables x_i's (resp. y_j's) and the target values $\mathbf{x}_i^k$ (resp. $\mathbf{y}_j^k$). We note $\mathcal{F}(\mathfrak{R}) = \{r_k | k \in \mathfrak{R}\}$ the set of rule utility functions.

Example 5 (Scene rule utility)**.** *Let us consider rule (3.1) from example 3, where s_1 is the value of the presence sensor. Here a possible utility function, which is the negated* distance *between the current value of y_1 and the target value $\mathbf{y}_1^1 = 60$ defined in rule (3.1):*

$$r_1(y_1, s_1) = \begin{cases} -|y_1 - 60| & \textit{if } s_1 = 1 \\ 0 & \textit{otherwise} \end{cases}$$

Here a possible utility function for rule (3.3), where s_2 is the sensed light level and x_3 is the level of the shutter:

$$r_1(y_1, x_3, s_1, s_2) = \begin{cases} -\sqrt{|y_1 - 60|^2 + |x_3|^2} & \textit{if } s_1 = 1,\ s_2 > 60 \\ 0 & \textit{otherwise} \end{cases}$$

Using these notations, we can express the **SECP** as an optimization problem. Our goal is to maximize the utility of the user-defined rules, while at the same time minimizing the energy consumption of actuators, which can be written as follow:

$$\underset{x_i \in \mathcal{X}(\mathfrak{A})}{\textbf{minimize}} \sum_{i \in \mathfrak{A}} e_i \quad \text{and} \quad \underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \sum_{k \in \mathfrak{R}} r_k \tag{3.5}$$

Of course, this formulation uses the scene action variables y_i, whose values cannot be modified directly (i.e. y_i's are not decision variables in our problem) but can be predicted using our physical

model functions, based on the actuators' variables. This introduces a new set of constraints

$$\phi_j(v_j^1, \ldots, v_j^{\overline{\phi_j}}) = y_j \qquad \forall y_j \in \mathcal{X}(\Phi) \tag{3.6}$$

As physical models represent the real world physical constraints they are not something we can actually optimize and we must model them as hard constraints.

Based on this, we can straightforwardly map the **SECP** to a multi-criteria optimization problem.

$$\begin{gathered} \underset{x_i \in \mathcal{X}(\mathfrak{A})}{\textbf{minimize}} \sum_{i \in \mathfrak{A}} e_i \quad \text{and} \quad \underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \sum_{k \in \mathfrak{R}} r_k \\ \textbf{subject to} \;\; \phi_j(v_j^1, \ldots, v_j^{\overline{\phi_j}}) = y_j \quad \forall y_j \in \mathcal{X}(\Phi) \end{gathered} \tag{3.7}$$

Given all the previous concepts and notations, we define the **SECP** as follows:

Definition 3 (Smart Environment Configuration Problem (SECP)). *Given*

- *a set of actuators $\mathfrak{A}$, and their related costs $e_i \in \mathcal{F}(\mathfrak{A})$,*
- *a set of sensors $\mathfrak{S}$,*
- *a set of scene rules $\mathfrak{R}$ and their related utility functions in $r_k \in \mathcal{F}(\mathfrak{R})$,*
- *a set of environment states Φ, and a set of physical dependency models $\mathcal{F}(\Phi)$,*

the Smart Environment Configuration Problem *(or* SECP*) is represented by a tuple $\langle \mathfrak{A}, \mathcal{F}(\mathfrak{A}), \mathfrak{S}, \mathfrak{R}, \mathcal{F}(\mathfrak{R}), \Phi, \mathcal{F}(\Phi) \rangle$ and amounts to finding the configuration of actuators that maximizes the utility of the user-defined rules, whilst minimizing the global energy consumption and fulfilling the physical dependencies.*

3.2 Solving the SECP with a DCOP approach

As we have seen in the previous section, we can model the configuration problem in a smart environment as an optimization problem. In this section, we show how this optimization problem can be solved in a distributed setting such as **AmI** scenarios. As the smart environments can naturally be represented as a **MAS**, we map our **SECP** optimization problem to a **DCOP**.

3.2.1 Mapping the SECP to a DCOP

In the previous section, we described the problem of the **SECP** as a multi-objective optimization problem. We will now introduce how this optimization problem can be mapped to a **DCOP**.

Our **SECP** is currently represented as a multi-objective optimization problem 3.7, which is not convenient when mapping to a **DCOP** We could use a **MO-DCOP** (see 2.3.2.3) but solution methods for this **DCOP** extension are quite heavy and would not fit our target environment, composed of constrained devices. Instead, we choose to aggregate the two objectives to formulate

the problem as a mono-objective optimization problem, using weights $\omega_u, \omega_c > 0$:

$$\begin{aligned} \underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \quad & \omega_u \sum_{k \in \mathfrak{R}} r_k - \omega_c \sum_{i \in \mathfrak{A}} e_i \\ \textbf{subject to} \quad & \phi_j(v_j^1, \ldots, v_j^{\overline{\phi_j}}) = y_j \quad \forall y_j \in \mathcal{X}(\Phi) \end{aligned} \tag{3.8}$$

As described in 2.3.2, a DCOP is represented by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$, therefore to map the **SECP** to a **DCOP** we need to define the sets of agents $\mathcal{A}$, variables $\mathcal{X}$, domains $\mathcal{D}$ and constraints $\mathcal{F}$ and the mapping function μ.

3.2.1.1 Agents

In a **DCOP**, agents control decision variables and are responsible for selecting an appropriate value for each of the variable they own. This also means that agents are the entities that perform any computation required by the **DCOP** algorithm. In the case of a physical distributed system like the one described in our **AmI** scenario, agents must be embodied by real devices, which must possess some processing and communication capabilities.

The various connected devices, sensors and actuators, available in our system fit these requirements and thus can be considered as agents. However, these devices are assumed to be resources constrained and the communication link between them is generally implemented with a low power network. Devices with only a sensing role are usually powered on battery and run as *sleepy nodes*, meaning that they switch off their communication interface most of the time to save energy and only turn it on when they want to emit a new value. Their processing power is also severely limited due to these energy saving constraints. On the other hand, actuator devices, for example light sources, are usually connected to the main power line and always reachable. It should be noted that while many real-life products embed both sensors and actuators, sensing-only devices are really powered on battery in most cases. For simplicity, and without loss of generality, we consider our devices to have a single role, either actuator or sensor. Therefore, our set of agents is made of the actuator devices and $\mathcal{A} = \mathfrak{A}$. Agent with actuator i is denoted a_i and the set of agents correspond to $\mathcal{A}$.

3.2.1.2 Variables and Domains

The variables used in our **DCOP** is simply the set of variables used in the multi-objective optimization problem 3.7 representing the **SECP**. More precisely, the set $\mathcal{X}$ contains all the variables whose values are selected by an agent:

- Actuator variables x_i can clearly be controlled by agents and are part of $\mathcal{X}$
- Scene action variables y_i, which represents predicted states of the environment, are also part of x. These variables do not represent any action or decision in the physical environment, they are *modeling variables*. Yet, agents do try to affect them a value that reduces the distance to rule's goal (i.e. increase its utility) and $\mathcal{X}(\Phi) \subset \mathcal{X}$.
- On the other hand, the sensor variables s_l are not part of $\mathcal{X}$ as these variables represent sensor values that cannot be controlled by an agent.

This gives us the following definitions:

$$\mathcal{X} = \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi) \tag{3.9}$$

$$\mathcal{D} = \{\mathcal{D}_{x_i} | x_i \in \mathcal{X}(\mathfrak{A})\} \cup \{\mathcal{D}_{y_j} | y_i \in \mathcal{X}(\Phi)\} \tag{3.10}$$

3.2.1.3 Constraints

Constraints of the **DCOP** are obviously based on the constraints of the optimization problem 3.7. However, this problem has a mix of hard and soft constraints, which cannot be dealt with directly by **DCOP** algorithm, and **DCSP** only support hard constraints. Therefore we need to encode the hard constraints 3.6 as soft constraints with infinite costs, noted y_j for each $j \in \Phi$:

$$\varphi_j(x_j^1, \ldots, x_j^{\overline{\phi_j}}, y_j) = \begin{cases} 0 & \text{if } \phi_j(x_j^1, \ldots, x_j^{\overline{\phi_j}}) = y_j \\ -\infty & \text{otherwise} \end{cases} \tag{3.11}$$

We note the set of translated hard constraints $\mathcal{F}(\mathbf{\Phi})$ and the set of constraints of the **DCOP** can be defined as:

$$\mathcal{F} = \mathcal{F}(\mathfrak{A}) \cup \mathcal{F}(\mathfrak{R}) \cup \mathcal{F}(\mathbf{\Phi}) \tag{3.12}$$

3.2.1.4 Full DCOP Definition

Using these definitions, **SECP** is then formulated as a **DCOP** $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ where μ is a function that maps variables and constraints to agents; with the following objective:

$$\underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \quad \omega_u \sum_{k \in \mathfrak{R}} r_k - \omega_c \sum_{i \in \mathfrak{A}} e_i + \sum_{j \in \Phi} \varphi_j \tag{3.13}$$

3.2.2 Factor Graph Representation

The DCOP (3.13) contains many non-binary constraints, for physical models, and thus the traditional constraint graph representation is not really convenient is this case. Instead we represent the DCOP modeling our SECP using a factor graph (see 2.3.1.1). This representation allows to clearly visualize the spatial relationships between physical models, actuators and sensors (see Figure 3.7). Additionally, it is used as a basis for several **DCOP** solution methods.

Actuators are represented in the factor graph by pairs made of a factor vertex and a variable vertex, as displayed by Figure 3.2. The variable node maps to the variable x_i representing the state of the actuator, while the factor node represents the cost function for this actuator and is a unary constraint.

Physical models are also represented as (factor, variable) pairs, where the variable represents the expected state of the environment, and is linked to active rules setting a target for this state. The

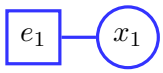


Figure 3.2 – Factor graph actuator representation

factor vertex represents the hard constraint φ_i used to ensure that this estimated state maps the value returned by the physical model function based on the actuator value and is linked to the variable vertices from actuators influencing this state.

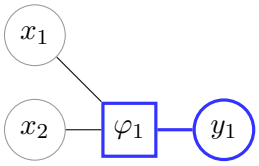


Figure 3.3 – Physical model representation in a factor graph

Sensors are only used to trigger rules in the **SECP** and are not involved in the DCOP optimization process, which only occurs when the set of active rules has been determined. Therefore, sensors do not need to be represented in the graph. moreover they could not be represented in a classical factor graph, which is a bipartite graph and thus a only two kind of vertices. However, when it helps comprehension, we will represent them on the graph using diamond-shaped vertices.

Figure 3.4 – Sensor representation in the factor graph

Rules are represented by single factors, which represents utility functions r_k of the rules. Such a factor is linked to the variable(s) the rule set a goal on, (either x_i's or y_j's) and to the variables used in the condition part of the rule

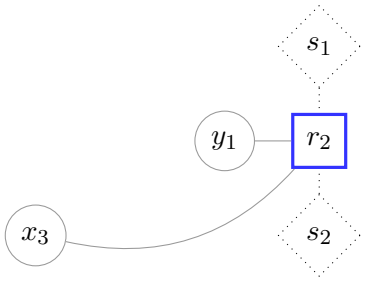


Figure 3.5 – Rule representation in the factor graph

Example 6. *For example, Figure 3.6 represents a factor graph for the DCOP of a simple DCOP from Example 3, with 3 actuators (the 3 light bulbs), one physical model and environment state (the light level in the living room), two sensors (presence and luminosity) and one rule.*

Example 7. *Using these notation, we can also represent the factor graph for a complete house level. Figure 3.7 depicts the Factor Graph for the SECP of a real house level (actually it is the*

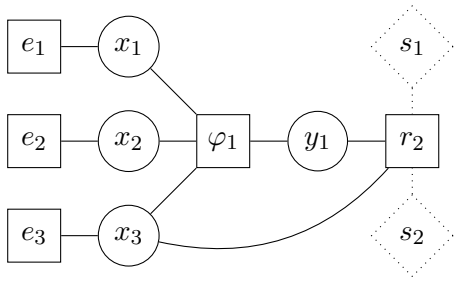


Figure 3.6 – Factor graph for the scenario of Example 3

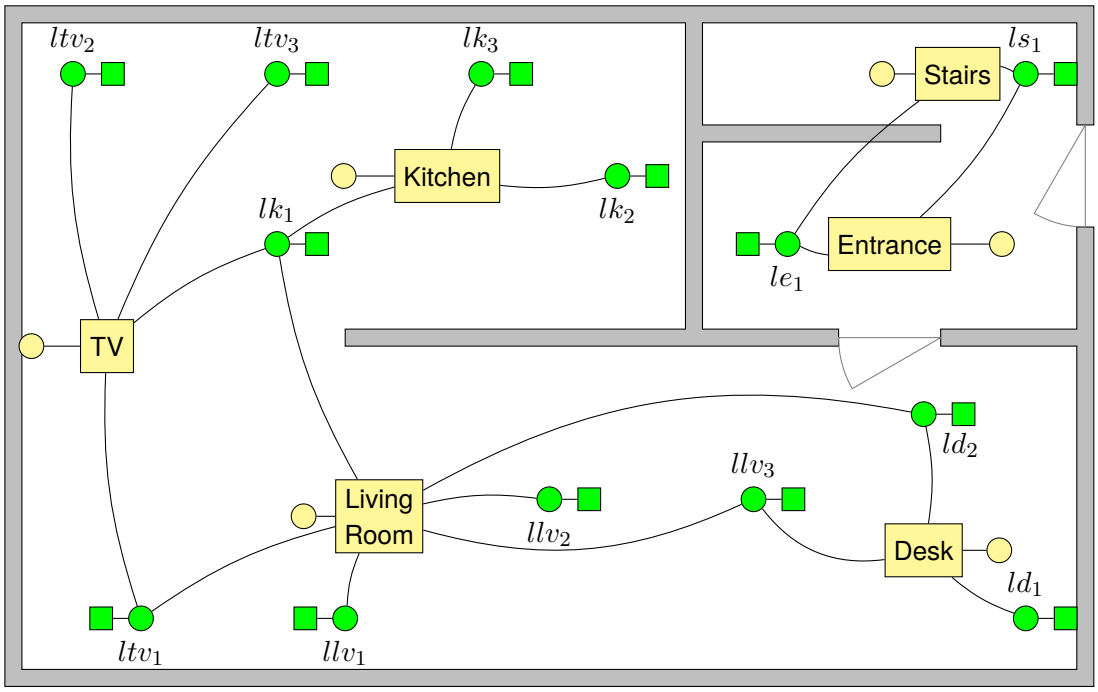


Figure 3.7 – Factor graph for a realistic full house level

author's house!). Note that, for clarity, rules have been omitted from this figure.

3.3 Experimental Evaluation

In order to validate the **SECP** model, and the applicability of **DCOP** solvers, we evaluate it on simulated environments and use several **DCOP** algorithms to solve it.

3.3.1 Experimental Setup

We consider a realistic smart home with actuators (light bulbs), physical models and user-defined rules. Notice that, for simplicity sake, we only consider light control in our experiments, even though our model could be applied to other parameters in a house.

As presented previously, each actuator is represented by a variable x_i associated with an efficiency factor e_i, which defines a cost function as a linear function of the emitted luminosity. Each physical dependency model is represented by a pair (φ_j, y_j), where φ_j is defined as weighted sums (weights are randomly selected) of the luminosity levels emitted by the light bulbs in its scope and yield the

theoretical resulting luminosity in a given place as an indirect scene action variable y_j. Finally, each rules r_k assigns target values to one or several scene action variables (actuators and models). Variables, models and rules are randomly connected and we only consider active rules, which have an actual influence on the problem. We use $\omega_c = 1$ and $\omega_u = 10$ as weights when aggregating the two objectives (respectively for energy cost and rules utility).

Notice that the resulting **DCOP** is distributed using the **GH-SECP-CGDP** or **GH-SECP-FGDP** algorithms, depending on the graphical model used by the algorithms selected to solve the problem. These distribution methods are presented in Sections 4.3.1 and 4.3.2 respectively.

Each instance is then solved using a set of suboptimal **DCOP** algorithms, which are parametrized as follow:

- **DSA** [157], a stochastic local search algorithm. We use the **DSA**-B variant with a probability of parallelism $p = 0.7$.
- **MGM** [77], which has no specific parameter.
- **MGM-2** [77], a 2-coordinated variant of **MGM**. We use $q = 0.5$ as a threshold for becoming an offerer.
- **A-MaxSum** [36], an asynchronous implementation of the **MaxSum** algorithm. We use a damping [23] factor of 0.2, applied both at factors and variables nodes and add noise to energy cost constraints for easier tie breaking. Initial messages are sent at startup both from leafs in the graph and variables.

We have selected these algorithms because they are very lightweight and fast and exchange generally small messages, meaning that they should be ideal in our target environments made of constrained devices with limited communication. Moreover, they are known to generally produce good quality solutions.

We also solved the very same instances with a complete algorithm, **DPOP**. This gives us a reference optimal cost to evaluate the quality of the solutions produced by other algorithms (if and when **DPOP** is able to find it in a reasonable time). When using this complete algorithm (**DPOP**), each instance is solved once with a 120-second time limit When using a sub-optimal algorithm (**A-MaxSum**, **MGM MGM-2**, **DSA**), instances are solved 10 times with a 10-second time limit. pyDCOP implementation (see Section 6) is used for all algorithms.

3.3.2 Increasing House Size

In this first experiment, we progressively increase the number of lights, physical models and rules in the system, which represents progressively larger houses.

Each physical model is randomly connected to 1 to 4 lights and each rule is randomly connected to 1 to 3 models or lights.

The smallest instances have 10 lights, 3 models and 2 rules, and the count of each of these elements is linearly increased, by increments of 10 lights, up to 90 lights, 27 models and 18 rules. For each problem size, 100 instances are generated and solved (900 in total). Overall, 36900 algorithms executions are run for this experiment and results are averaged.

3.3.2.1 Solving the Instances

The first interesting thing to notice is that **DPOP** does not always succeed in producing a result in the 120-second time limit, even though we allowed it a much bigger timeout than for suboptimal algorithms.

Table 3.1 lists the rate of failure when solving our instance with **DPOP**. We can clearly see that **DPOP** struggles with bigger problems, where it fails on more than a third of the instances. This can be explained by the fact that **DPOP** complexity is $O(d^w)$, where w is the induced width of the pseudo-tree and d is the size of the largest domain. As we generate our instances randomly, we don't have a fixed width for the pseudo-tree and, when generating large instances, the resulting pseudo-tree width also tends to be larger, which can easily lead to reaching the timeout when computing the cost hypercube for an agent.

On the other hand, the suboptimal algorithms we are using here can be interrupted at any time and thus always produce a result. We consider for now the result as produced at the end of a 10-second timeout.

In the remainder in this section, figures will only report results for instances that were successfully solved.

# lights	% of failure
10	0 %
20	0 %
30	2 %
40	2 %
50	1 %
60	16 %
70	17 %
80	34 %
90	43 %

Table 3.1 – Failed SECP instance when using **DPOP**

3.3.2.2 Hard Constraints Violations

The **SECP** model contains a mix of soft constraints (for rules utilities and actuators' energy costs) and hard constraints (for physical models), which is a difficult situation for many **DCOP** algorithms. Therefore, we analyse here the number of hard constraints violated by each algorithm when solving our instances.

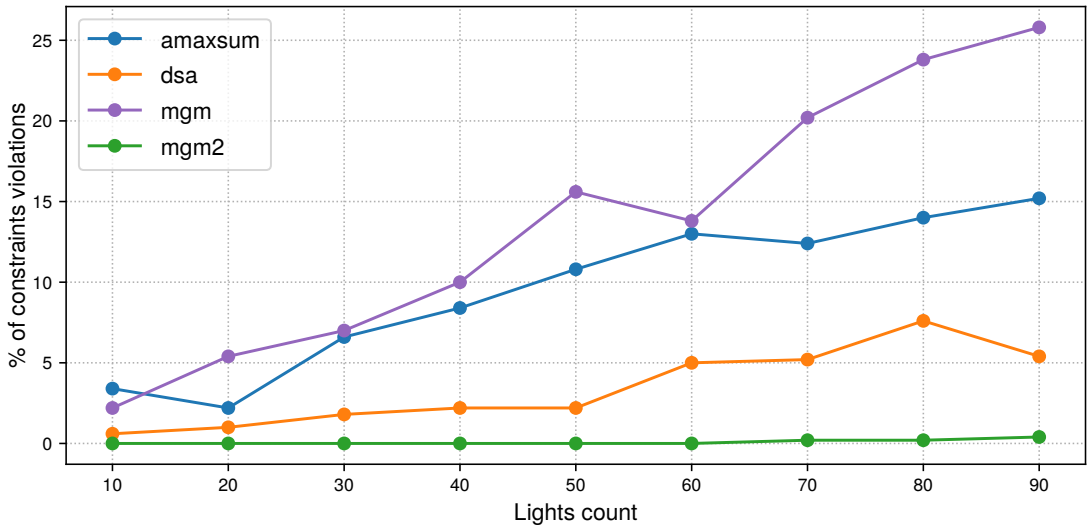


Figure 3.8 – % of constraints violation for increasing size **SECP** instances with several **DCOP** algorithms

Figure 3.8 shows the percentage of executions, across all instances, for which the solution was found to violate a hard constraint in the **DCOP**. All our problems are feasible, thus **DPOP**, when it succeeds in solving the problem, never violates any hard constraint and is not depicted here.

We can see that **MaxSum** tends to yield a large number of violations when the problems get bigger, up to 15% when using 90 lights. We have tuned the algorithm's parameters to avoid this phenomena, especially damping and noise level. While this helps reducing it a lot (we got up to 40% of violations on simple instances before tuning) we could not achieve better results with **A-MaxSum** and the **SECP** problem definition. However, when studying the results produced by **A-MaxSum**, we can see that most violations are due to a difference of 1 on the selected value for one light variable, meaning that it would generally not be perceptible by end users. Moreover, we believe these results could be further improved by a better handling of infinite costs for hard constraints or by decimating [20] variables involved in these hard constraints.

MGM also tends to produce a large number of constraints violations, but for different reasons. As it is monotonous and only allows a single variable to change it's value, **MGM** gets very easily trapped in local optima, especially with problems like the **SECP** where two variables must be changed simultaneously to obtain a gain in cost. Unlike **MaxSum Algorithm (MaxSum)**, we observe that the solutions produced by **MGM** that contains violations can be arbitrary bad: once a local optima has been reached it has no way of escaping it.

This can be easily seen by looking at results produced by **Maximum Gain Message with 2-coordination (MGM-2)**, the 2-coordinated variant of **MGM**, which never broke any hard constraints in our experiment. **MGM-2** always considers the hard constraints first, as solving these constraints yields the maximum gain. And as it is monotonous, once the hard constraints are satisfied, it will never select a value that would break them when optimizing for the soft constraints.

Finally, **DSA** exhibits a low number of violations (approximatively 5%), which does depends on the size of the problem.

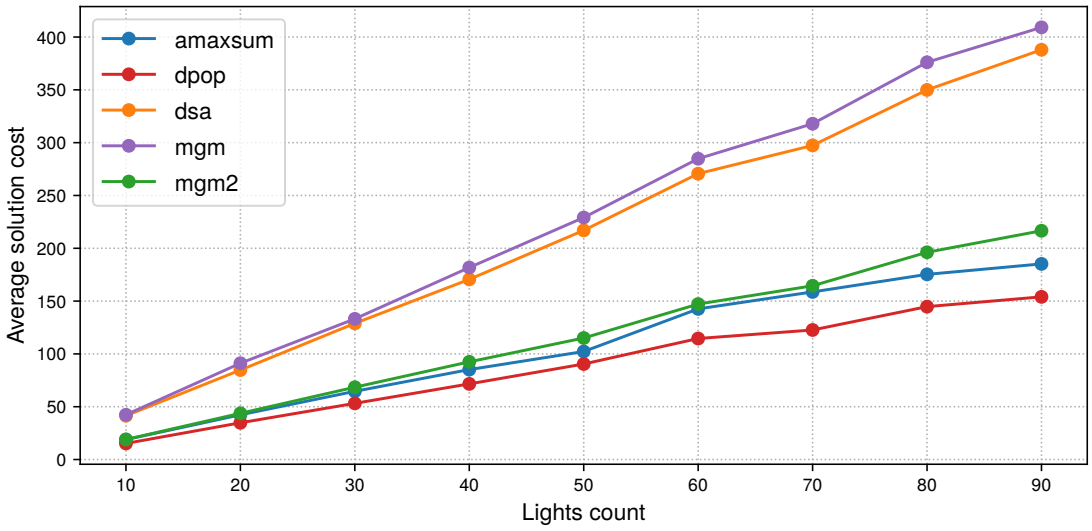


Figure 3.9 – Solution costs for increasing size **SECP** instances with several **DCOP** algorithms

3.3.2.3 Solutions Quality

Figure 3.9 shows the average cost of the solution found by each of the algorithms, only accounting for instances that have been actually solved and that did not violate any hard constraint.

As **DPOP** is complete, it always produces the lowest cost, which we can use to evaluate the quality of the solutions from other algorithms. We can see that **MGM-2** and **A-MaxSum** produce very good quality results. Given the fact that **DPOP** could not solve many of these problems, we argue that **MGM-2** and **A-MaxSum** are good candidates for solving **SECP**.

On the other hand, **MGM** and **DSA** produce similar results of relatively poor quality.

3.3.2.4 Execution Time

When solving the instances we allocated a 10-second time budget to suboptimal algorithms, however, most of the time these algorithms actually find their solution and stop changing their assignment much faster than this. In order to fairly compare the time at which each algorithm could really deliver an assignment for the environment configuration, we now look at the time at which each algorithm stopped changing its result.

Figure 3.10 plots these durations for each algorithm and each problem size. As we can see **MGM** and **DSA** are the fastest algorithms on our problems, and exhibit a remarkable stable resolution time as the problems grow. However, this is due to the fact that they rapidly get trapped in local minima, as we have seen on Figure 3.9: their speed is actually achieved at the cost of their results' quality.

A-MaxSum has relatively fast execution times. Given that it also has good solutions quality, this reinforce our opinion that this algorithms is well suited for configuring **AmI** environment using the **SECP** model.

On the other hand, based on Figure 3.10 **MGM-2** seems to be quite slow compared to the three previous algorithms. This is however not entirely true: Figure 3.11 shows the evolution of the cost

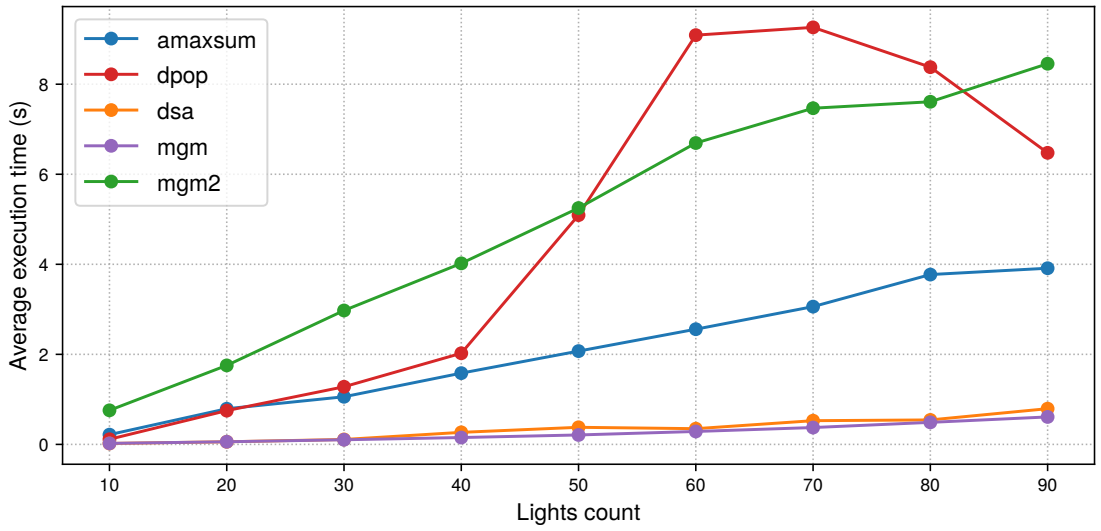


Figure 3.10 – Execution time for increasing size **SECP** instances with several **DCOP** algorithms

of the solution output by **MGM-2** on a typical large instance (90 lights) with a zoom, on the right, on the behavior of the algorithm after 2 seconds. As we can see, **MGM-2** produces a very good results after 4 seconds, but still makes very small improvements up to after 7 seconds, which is thus the time that is plotted on Figure 3.10. Therefore, we argue that **MGM-2** has actually very good response times.

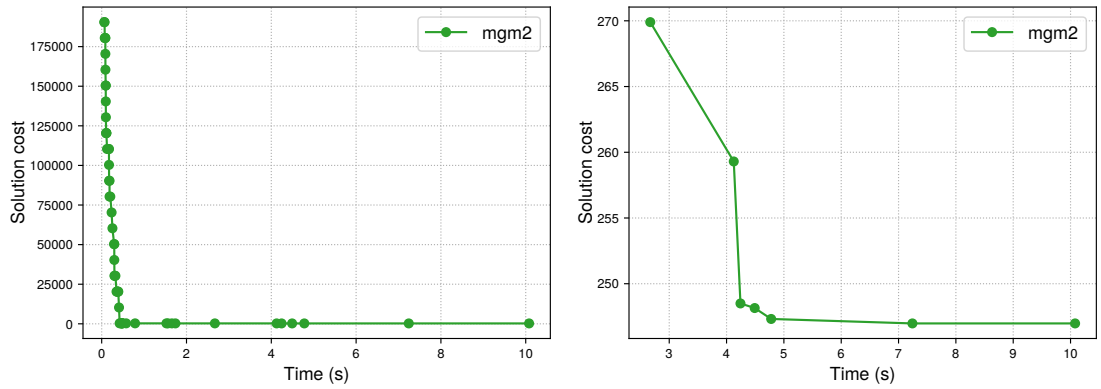


Figure 3.11 – Solution cost over time for **MGM-2** on a large **SECP** instance

Finally, **DPOP** exhibits the slowest performance on large problems, with a large variation across instances. The situation is actually even worse than what is depicted on Figure 3.10 as instances that could not be solved within the 120-second time budget are not taken into account on this plot, which explains the decrease for the three largest instances sizes (a large proportion of the most difficult instances could not be solved, hence we only plot the easiest once). **DPOP**'s complexity induces very large computation on nodes with many pseudo-parents in the tree, which explains these results.

3.3.2.5 Impact on Communication

Finally, as communication protocols typically used **AmI** environments have limited bandwidth, we also evaluate the communication load induced by each of these algorithm when solving the **SECP** instances. We define the communication load as the sum of all messages exchanged by all agents when solving the problem. Figure 3.12 plots the average number of messages and the average communication load for each algorithm and each problem size.

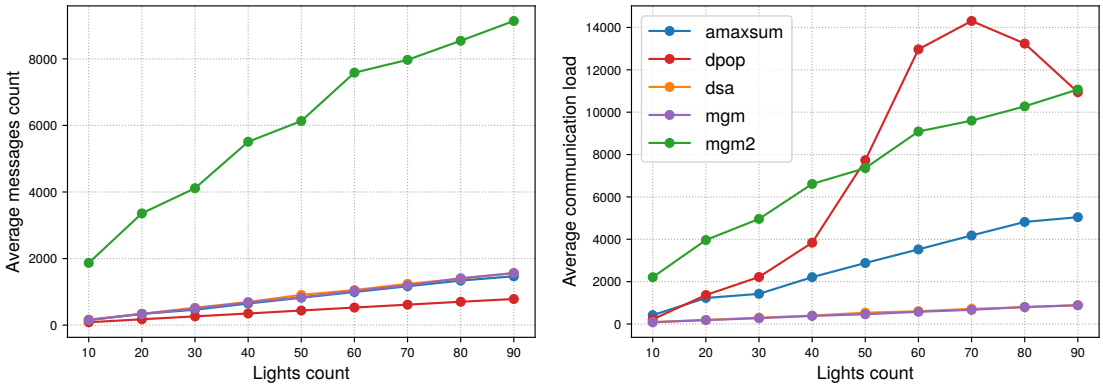


Figure 3.12 – Messages count and communication load for increasing size **SECP** instances with several **DCOP** algorithms

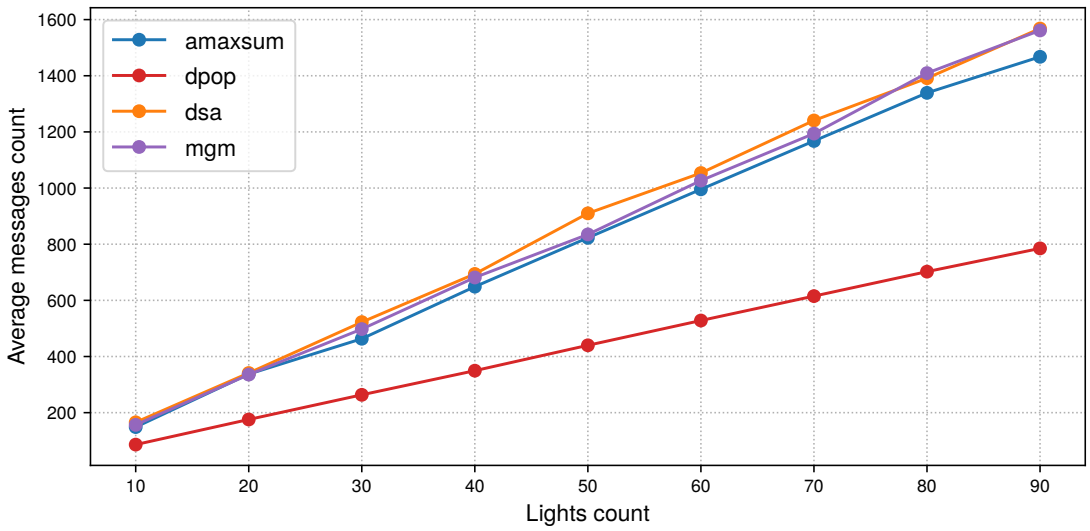


Figure 3.13 – Messages count for increasing size **SECP** instances with several **DCOP** algorithms

When looking at the messages count, all algorithms seem to behave similarly, except **MGM-2**, whose coordination mechanism generates a very large number of messages, which could prove to be a problem on systems where latency might be high. However, this load should be minored, as a large number of messages are due to very small improvements made during the second half of the 10-second time budget, as it has already been depicted for cost on Figure 3.11.

In order to compare other algorithms, we must look at Figure 3.13, which excludes **MGM-2**. Of course **DPOP** has a very low number of messages, as each node only sends very few messages: one `VALUE` and one `COST` message during the optimization process and at most one to each neighbor

for building the **DFS Tree** tree. Notice that, as previously, the figure for large instances does not represents the reality of all instances, as many could not be solved. The last three remaining algorithms present a reasonable number of messages, with **A-MaxSum** being more frugal than **DSA** and **MGM**.

When looking at communication load, things are different: despite generating few messages, **DPOP** generally has an high communication load, which seems to be correlated with the large execution time we observe on Figure 3.10: as a matter of facts nodes with a large number of pseudo-parents must generate, and send, very large cost hypercubes.

As for execution time, **DSA** and **MGM** have approximatively the same, very small, communication load, which is due to the same reasons: as they get trapped in a local optima they quickly stop exchanging messages and achieve small communication load and execution time, but poor quality results.

We can also see that **MGM-2** exhibits a relatively large network load, although not as high as the message numbers could have suggested: indeed **MGM-2**'s coordination messages are numerous but very small. Finally, **A-MaxSum** generates reasonable network load.

3.3.3 Increasing House Complexity

In this second experiment we generate problems with an increasing number of rules, for the same number of physical models and lights. All our instances have 30 lights and 10 models and the number of rules ranges from 1 to 9.

As previously, we generated 10 instances for each problem size and solved each of these instances 10 times with each suboptimal algorithms and once with **DPOP**.

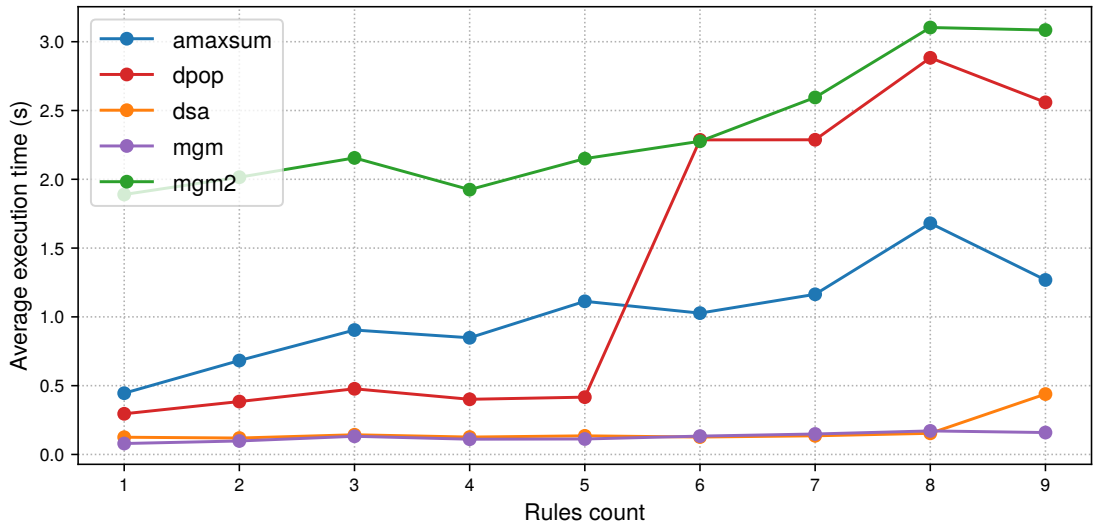


Figure 3.14 – Average execution time for **SECP** with a growing number of rules, solved with several **DCOP** algorithms

We can see on Figures 3.14, 3.15, and 3.16, that the results are very similar to what we observed when increasing the number of lights, models and rules.

Interestingly, we can observe that increasing rules density does not make the problem excessively harder to solve, which means that the **SECP** model is able to cope with complex environments with a relatively large number of rules. The main differences with the experiments from Section 3.3.2 is that no hard constraint was violated by any algorithms in these settings: it seems that hard constraints violation depends on the problem size more than on its density.

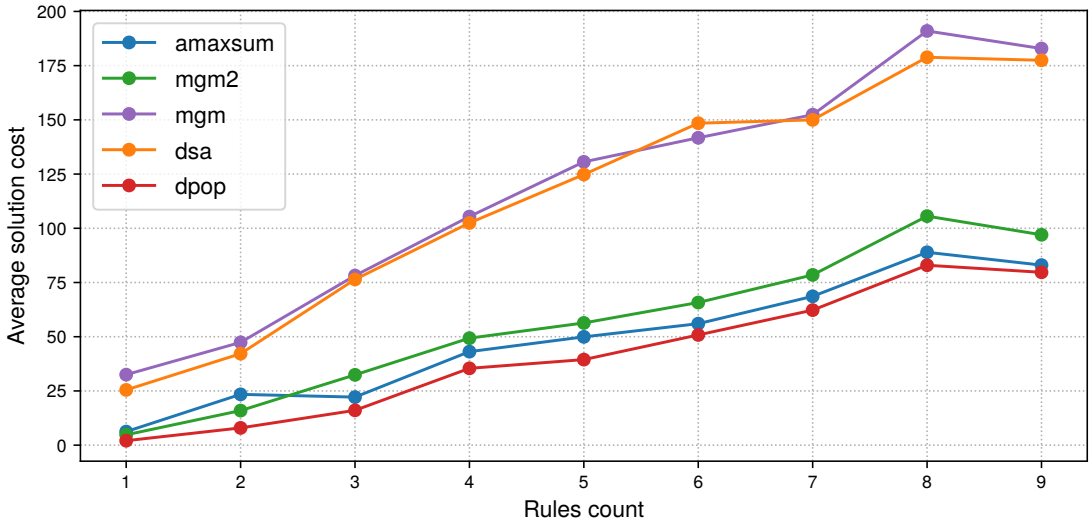


Figure 3.15 – Average solution cost for **SECP** with a growing number of rules, solved with several **DCOP** algorithms

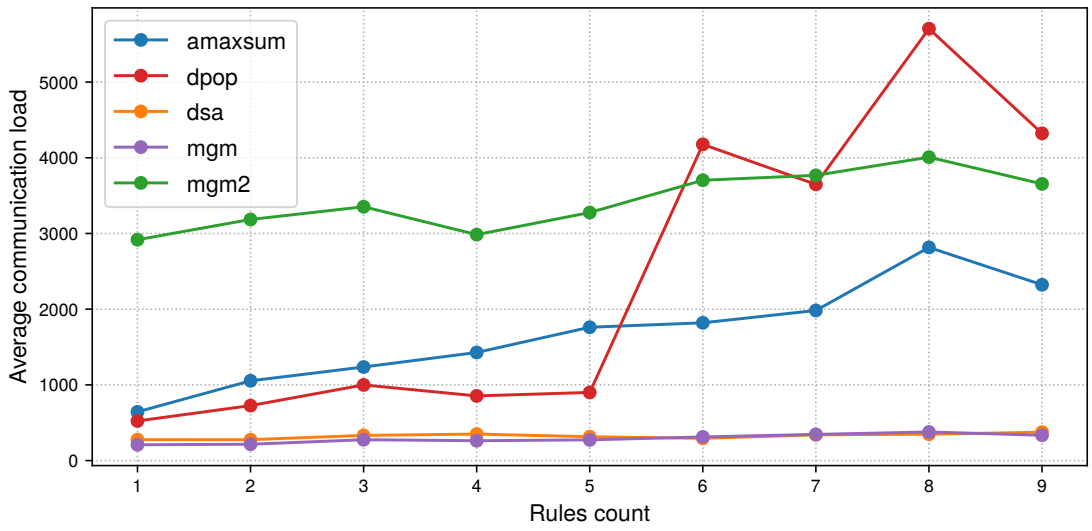


Figure 3.16 – Communication load for **SECP** with a growing number of rules, solved with several **DCOP** algorithms

3.3.4 Conclusion of Experimental Evaluations

From these experiments on simulated smart home environments, we can see that the **SECP** model, when mapped to a **DCOP** can be used to implement coordination among the devices in environments.

We tested these problems with several off-the-shelf **DCOP** solution methods, which allows us to draw the following conclusions:

- **DPOP** is not well suited for this use case: while it produces optimal solutions, its execution time and communication load are prohibitive.
- **MGM** and **DSA**'s response time and communication load make them good candidates, however, their tendency to be trapped in local optima and the resulting low quality of the solutions they produce is problematic.
- **MGM-2** is fast, produces good quality results and never violates any hard constraints. However the communication load it induces is too high and the number of messages it generates might be problematic, especially if the network has a high latency.
- **A-MaxSum** seems ideal, providing good results in reasonable time with a modest network load. Moreover, its performance are remarkably stable as the problems grow in size and complexity. However, the number of violations of hard constraints might be problematic.

As a consequence, we argue that suboptimal belief-propagation based algorithms like **MaxSum** are best suited for these environments, but that variants of these algorithms should be developed to better handle a mix of soft and hard constraints, which is a common situation when modeling real world problems. Besides, we have only been using a standard 'off-the-shelf' version of **MaxSum**, it would be useful to develop a variant specifically tailored for **SECP**, for example by using domain pruning based on the model's specific structure and characteristics, to mitigate these issues. For large problems, cutting edges or decimating variables [20] could also help to reduce them to smaller problem, which suffer less from hard constraints violation, as we have seen in our second experiment. We keep these research directions as perspectives.

3.4 Summary

In this chapter, we have proposed a model for goal-oriented coordination among connected devices in a Smart Environment. Devices operate themselves the configuration process, without supervision. The model makes use of physical relations between objects as to prevent the user to explicitly specify the role of each object, easing the definition of rules and the introduction of new devices at runtime.

We have shown that this model could be mapped to a **DCOP** and propose to use message-passing methods to implement the coordination protocol. From our experiments on simulated smart home scenarios, suboptimal belief-propagation algorithms like **MaxSum** are best suited for the constrained devices commonly used in smart environment and our **SECP** model is a viable approach for autonomous coordination among these devices.

In the next chapter we will tackle another challenge for applying **DCOP**-based model to real-world situations by studying how this model can be deployed on physical devices in the house.

4 Distributing Decisions

In Section 2.3.2 we stated that most works are based on the assumption that there is a one-to-one mapping between agents and variables in a **DCOP**. In this section, we discuss the limits of this assumption when using **DCOP** on real-world problems and argue that, when abandoning it, we must solve a *distribution problem*, as to decide how to map variables, and possibly constraints, to the agents of the **DCOP**. We propose a definition for an optimal distribution and develop several solution methods for computing optimal and approximate distributions.

4.1 On the Need of Decision Distribution

In this section we argue that the commonly used assumption on **DCOP** distribution does not hold when working on real world problems and that it is necessary to consider the question of the distribution of decisions over a set of agents. While few works currently exists in this domain, we consider it to be an important part of using **DCOP** approaches in physical distributed environments.

4.1.1 Classical Representation and One-to-One Mapping Assumption

A **DCOP** is generally represented as a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$ where μ is a function that assigns the control of each variable to an agent: $\mu : \mathcal{X} \rightarrow \mathcal{A}$. The agent hosting variable x_i is given by $\mu(x_i)$ and we note $\mu^{-1}(a_a)$ the set of variables assigned to agent a_a.

The classical assumption is that μ is a one-to-one mapping (see Section 2.3.2.2), which makes the problem easier as one only has to reason on the variables and can practically ignore agents when designing an algorithm.

This assumption is motivated by the fact that it is always possible to reformulate an arbitrary **DCOP** D into a new one D' where there is exactly one agent for each variable, by introducing either new variables or new agents.

- When using the *decomposition* approach, one variable in D' is created for each agent of D. This variable encapsulates the sub-problem representing the variables initially assigned to the agent by μ in D.

- When using the *compilation* approach, virtual agents are created in D' so that each variable in D is assigned to exactly one agent. At run-time a virtual agent representing a variable x_i is assigned to the real agent responsible for x_i such that $a_k = \mu(x_i)$

These two transformations indeed yield a new **DCOP** D' where the assumption holds and μ' is a one-to-one mapping. However, in order to apply these transformations, the mapping μ must be completely defined in the original **DCOP**. This is generally assumed to be true: as a matter of fact μ is a part of the tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$ that represents the **DCOP** in the classical definition.

From a **MAS** point of view, this definition of μ makes sense; the variables in the **DCOP** model decisions required to reach the overall goal and the actions the agents perform in the environment are based on these decisions: therefore one must decide which decision each agent is able to make. This is the reason why μ is supposed to be a part of the problem's definition, and not something that one has to figure out when solving the problem.

However, when formulating a real world problem as a **DCOP**, μ is often not well-defined: several definitions of μ are generally possible for the same sets of agents, variables, domains and constraints. We give several reasons for this in the next sections.

4.1.2 Natural Assignment of Decision Variables

When implementing a **DCOP** in a distributed system, agents are embodied in physical objects (computers, connected objects, etc.) and the actions/decisions of these agents are modeled as variables.

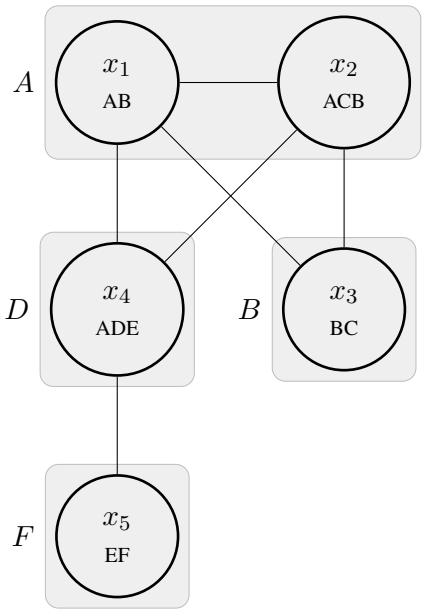
For example, in [35], Farinelli et al. model a power-constrained sensor-network using a **DCOP** where each variable represents the sleeping schedule of a device. In such a case, the assignment of these variables is obvious and they are assigned to the agent whose sleeping schedule they are modeling.

In general, a variable that models the decision of an agent *naturally* belongs to the agent that is making, and potentially applying to the environment, that decision.

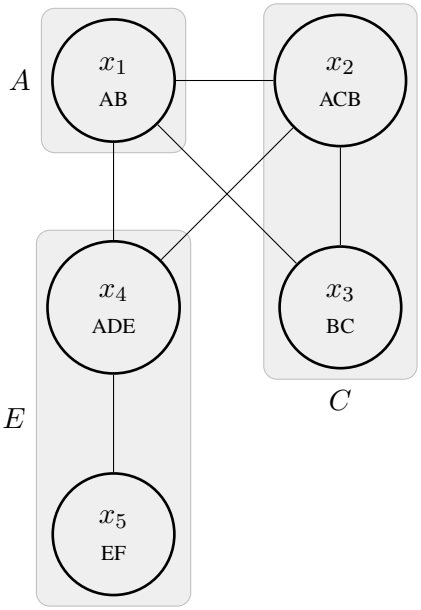
However it is common, when modeling a problem with a **DCOP**, to define variables that have no *natural* relationship to a single specific agent. As we will see in the next sections, these variables can either model *shared decision* or represent some abstract concept needed for decision (but without being the decision itself). One can think of this kind of variables as *modeling artifacts* or *auxiliary variables*. For these variables, the definition of μ is not directly given by the original real-world problem definition.

4.1.3 Shared Decision Variables

A common example of such a situation can be found in the *Event As Variable* (EAV) model for distributed meeting scheduling presented by Maheswaran et al. in [78]. In this model, agents represent the resources required for the meetings. Each meeting is represented by one variable in the **DCOP** and constraints are introduced to avoid overlap between two meetings that require the same resources. For example, if meetings E_1 and E_2, whose start times are modeled with variables x_1 and x_2, both require the resource A_1, a constraint $f_{1,2}$ between x_1 and x_2 ensures that the time



(a) Mapping on 4 agents, a variable is always assigned to an agent required for the corresponding meeting



(b) Mapping on 3 agents, a variable is always assigned to an agent required for the corresponding meeting

Figure 4.1 – With the EAV model, the same meeting scheduling problem with 6 resources and 5 meetings has multiple reasonable variable mappings

slots assigned to the two meetings do not overlap, in order to avoid conflicts. These variables are decision variables, but they model a *shared decision* among the agents/resources that participate in a meeting and could be reasonably assigned to any agent that takes part to that meeting, as stated by the authors.

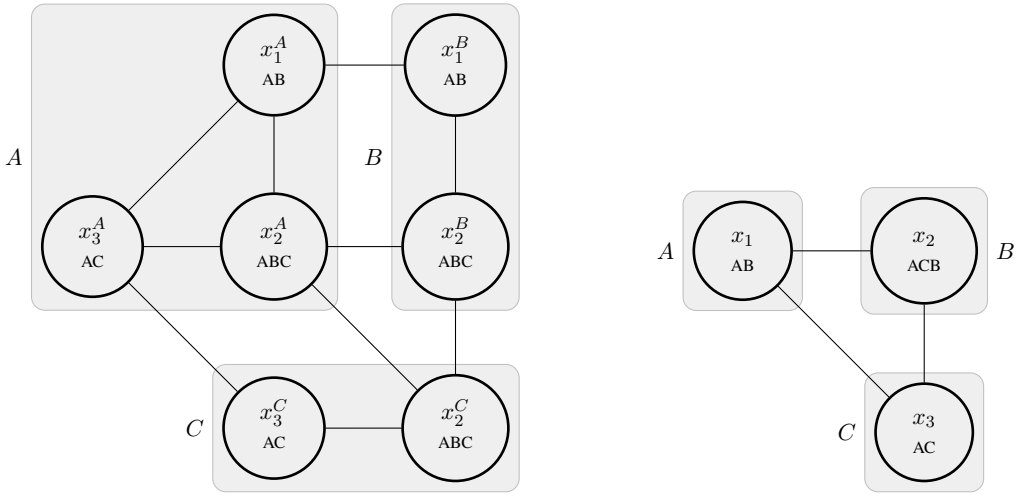
Example 8. *Figure 4.1 represents two possible definitions of the μ mapping function for a meeting scheduling problem with 6 resources $\{A, \ldots, F\}$ and 5 meetings $\{E_1, \ldots, E_5\}$ whose starting times are represented by variables $\{x_1, \ldots, x_5\}$. Each resource is represented by one agent. Resources required for each meeting are defined in table 4.1.*

Meetings	Resources / Agents					
	A	B	C	D	E	F
E_1	✓	✓				
E_2	✓	✓	✓			
E_3		✓	✓			
E_4	✓			✓	✓	
E_5	✓				✓	✓

Table 4.1 – Resources required for each meeting in a sample meeting scheduling problem

In both depicted mappings, the shared decision variables are always assigned to an agent that takes part in the decision; for example the variable x_3 can be assigned to either agent B (see Figure 4.1a) or C (see Figure 4.1b) as agents B and C represent resources that are both required for the meeting E_3 represented by x_3.

Another approach for the meeting scheduling problem, called *Private Event As Variable* (PEAV) is



(a) Mapping with the PEAV model

(b) One possible mapping with the EAV model

Figure 4.2 – Mappings, with the PEAV and EAV models, of the same meeting scheduling problem with 3 resources and 3 meetings.

also described in [78], where one variable is used for the decision of each agent and extra constraints are introduced to ensure that these local decisions are consistent with each other. This model avoids shared decision variables and exhibits a better respect of privacy. In PEAV, each variable belongs naturally to an agent (which explains the privacy advantages) and there consequently is only one logical μ mapping. However, the authors highlight that the EAV model outperforms PEAV by one order of magnitude, meaning that EAV would be a much better choice if strong privacy is not a prerequisite.

Example 9. *Figure 4.2a represents the mapping of variables to agents using the PEAV model for a meeting scheduling problem with 3 resources* $\{A, B, C\}$ *and 3 meetings* $\{E_1, E_2, E_3\}$ *Here a single meeting is represented by several variables, one for each of the resources taking part to that meeting.*

In contrast, the EAV model depicted in Figure 4.2b, requires only 3 variables, while 7 variables are needed when using PEAV.

In general, we argue that even if a problem can be modeled such that each variable belongs logically to a single agent, it is not necessarily a good option: depending on the requirements it might be better to use a simpler model with fewer variables, in which case the μ mapping might not be fully defined, as exemplified in the EAV model for meeting scheduling.

4.1.4 Auxiliary Variables

When modeling a complex problem, it is also common to introduce auxiliary variables in the model, which only serve as modeling artifacts but do not map directly to anything in the original problem.

As stated by Smith in [134], *"auxiliary variables are variables introduced into a model, either because it is difficult to express the constraints at all in terms of the existing variables, or to allow the constraints to be expressed in a form that would propagate better, i.e. lead to more domain*

reductions."

For example, when modeling a graph partitioning problem, one approach presented in [34] is to use binary variables $x_{ik} \in \{0, 1\}$ to indicate if x_i belongs to partition k. Then, variables $y_{ijk} = x_{ik} * x_{jk}$ are introduced and represent the fact that x_i and x_j belong to the same set k. While x_{ik}s can be seen as decision variables, y_{ijk}s are clearly auxiliary variables: they do not map directly to a decision and are only introduced to make the problem easier to model and/or solve (by linearizing the otherwise quadratic problem, in this case).

When mapping the **SECP** model, introduced in Section 3.2, to a **DCOP**, scene action variables y_j are also auxiliary variables and do not represent any agent's decision.

While common in general constraint reasoning, this kind of techniques is generally not used when modeling a problem as a **DCOP**. Most works use an approach that we could call an *agent-decision based model*: each variable only represents one single agent decision and the goal of the system is represented as the sum of individual agent's utilities (i.e. the social welfare), where each agent's utility is a function of the agent and its neighbors' decisions. Of course, in such a model, mapping variables to agents is straightforward.

Designing such a model is not always easy and generally leads to an high number of variables, as it requires avoiding any shared decision variable (as discussed in Section 4.1.3) and any modeling artifact variable. However, even when such a decision-based model is designed, it does not guarantee that the mapping of agents to variable will be fully defined, when applying the model to the real world, and implementing it on real devices.

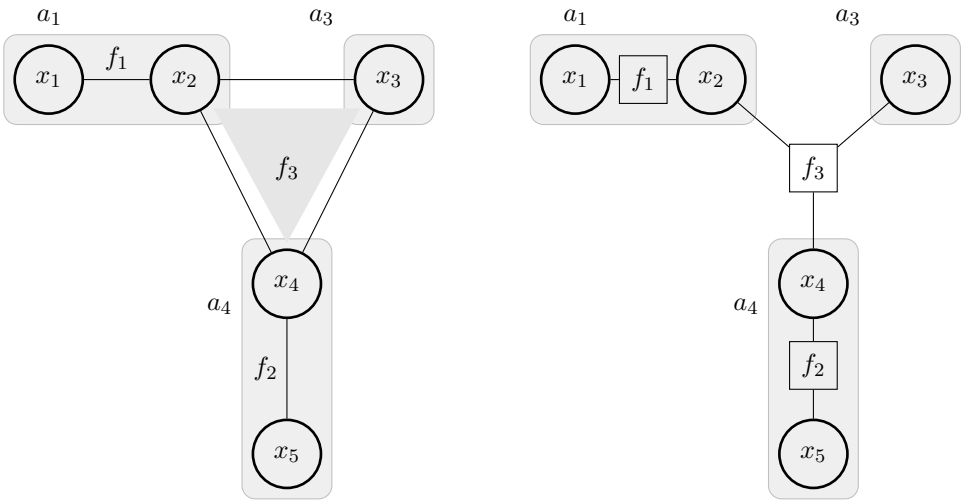
4.1.5 Binary-Constraints Assumption and Auxiliary Variables

Besides the one-to-one mapping, another generally used assumptions in **DCOP** works is that the problem only makes use of binary constraints (see Section 2.3.2.2). This means that for many **DCOP** algorithms, which only support binary constraints, the ideal decision-based model must be transformed into a **DCOP** with only binary constraints. Two classic methods are used for this transformation: the *dual graph method* and the *hidden variable method*. These transformations introduce new variables or constraints, whose mapping to agents is not obvious as we demonstrate in the following example.

Example 10. *Figure 4.3 represents the constraint graph and factor graph representations of a very simple* ***DCOP*** $D = \langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$ *with 5 variables, 2 binary constraints and one* 3*-ary constraint. In this* ***DCOP*** *the mapping* μ *is fully defined: each variable is assigned to exactly one agent.*

Binarization with the Hidden Variable Method. *When applying the hidden variable method,* D *is transformed into a new* ***DCOP*** $D' = \langle \mathcal{A}, \mathcal{X}', \mathcal{D}', \mathcal{F}', \mu' \rangle$ *represented on Figure 4.4. A new variable* x_{f_3} *is introduced by the binarization process, to represent the constraint* f_3*, along with 3 new binary constraints. As the constraints are not mapped to agents by* μ*,* x_{f_3}*, which replaces* f_3*, is also not mapped to any agent by the mapping* μ'*.*

Binarization with the Dual Graph Method. *Figure 4.5 represents the constraint graph of the* ***DCOP*** $D'' = \langle \mathcal{A}, \mathcal{X}'', \mathcal{D}'', \mathcal{F}'', \mu'' \rangle$*, obtained by applying the dual graph method to* D*. One*



(a) Constraint graph representation

(b) Factor graph representation

Figure 4.3 – A simple DCOP with a non-binary constraint

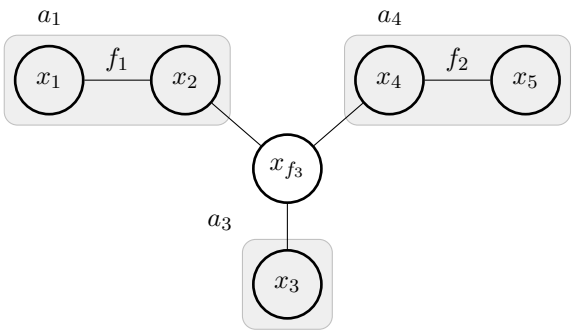


Figure 4.4 – Binarization with the hidden variable method

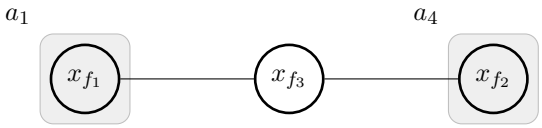


Figure 4.5 – Binarization with the dual graph method

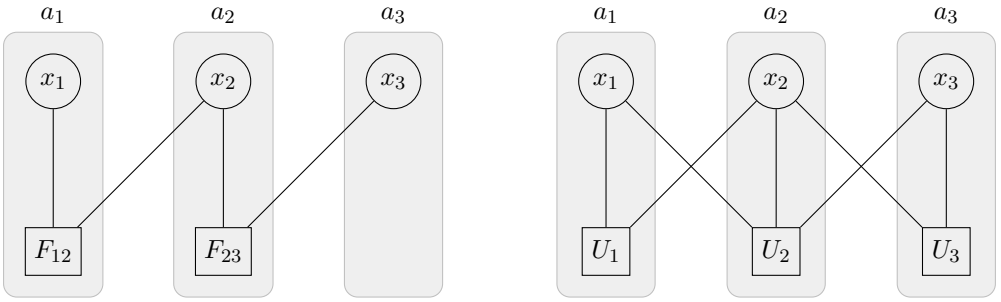
variable has been introduced in D'' for each constraint in D. As constraints f_1 and f_2 were only involving variables belonging to the same agent, we can reasonably assume that their mapping can be applied to the variables x_{f_1} and x_{f_2} in D''. However, the mapping of the variable x_{f_3}, which represents the constraint f_3 in D, is not defined.

As we can see from this simple example, when applying binarization to a **DCOP** the mapping of variables to agents introduces *auxiliary variables* whose assignment to agents is often not known, even though the original **DCOP** was decision-based, had not auxiliary variable and had a fully defined μ mapping.

4.1.6 Distribution of Factor Graph

In the classical **DCOP** definition, the mapping function μ assigns each variable to exactly one agent. However, algorithms based on a factor graph representation of the **DCOP** usually also require to assign constraints (a.k.a. factors) to agents. Indeed, as in these algorithms factor nodes also sends messages, these nodes must be allocated to agents, which are responsible for performing the computations required to generate such messages.

In [35], Farinelli et al. present two factor graph models for a sensor network coordination problem.



(a) Interaction-based factor graph

(b) Utility-based factor graph

Figure 4.6 – Two possible factor graph models for a sensor network coordination problem

In the first model, called *Interaction-based factor graph* and depicted on Figure 4.6a, the factors represent interactions between neighboring agents. To draw a parallel with the *shared decision variables* (see Section 4.1.3), one can think of these factors as *shared utility functions* (in the case of a maximization problem). In that model, the factor graph is a direct translation of the constraint graph, and the factors that represent the interaction have no obvious allocation to agents. On Figure 4.6a factor F_{12}, which represents the interaction between agent a_1 and a_2, could be assigned to any of these two agents.

The second model, called *Utility-based Factor Graph*, is depicted on Figure 4.6b. In this model, each agent has a function that represents its utility. This function is linked to all variables that

influence that utility. With this model there is a clear allocation of variables and constraints to agents: the μ mapping for variable is fully defined and each factor is also clearly assigned to exactly one agent. However, designing such a model is far from obvious and has consequences that might not be always acceptable. First, a utility-base factor graph requires to decompose the overall objective function into a set of agent utility functions. This decomposition is domain-specific, and generally harder to obtain than the interaction-based model. It's not even sure that it is always possible for an arbitrary problem. Furthermore, a utility-based model is generally not computationally efficient: it leads to a higher number of factors, which have a higher number of arguments, and creates loops in the factor graph. Factor graph-based algorithms usually struggle when dealing with loopy graphs. For instance, **MaxSum** is complete and optimal on acyclic graphs and approximate on loopy graphs, where it is not even guaranteed to converge. Given these issues, it might be beneficial to avoid graph with many cycles, even if that involves using a factor graph where some factors are not clearly allocated to a single agent.

As stated by the authors in [35], the choice of the factor graph representation is design choice that depends on the application requirements. Depending on these, it is common to end up with a **DCOP** definition where the mapping μ is not fully defined or/and where some factors are not clearly allocated to a single agent.

4.2 A Generalized Definition of Distribution for Deploying DCOPs

In this section we give a definition for the distribution problem for a **DCOP**. This problem amounts to finding a mapping of computations to agents, for a computation graph derived from a **DCOP**, when using a given **DCOP** algorithm.

As exposed in the previous section, for numerous reasons the mapping of variables to agents might not be obvious when using a **DCOP** approach on a real world problem. Furthermore, even when this mapping is available, it might not be enough for algorithms which also require allocating constraints to agents, and not only variables. In this section, we propose a generalized definition of this mapping, which we call a *distribution*.

4.2.1 Distributing Computations

As **DCOP** algorithms are distributed message passing algorithms, they are defined as a set of *building blocks*, where each block's behavior consists in sending and receiving messages. We call these building blocks *computations*, noted c_i:

Definition 4 (Computation)**.** *In a **DCOP** algorithm, a computation c_i is a piece of code that runs on an agent and only interacts with other computations through message sending.*

We note $\mathcal{C}$ the set of computations required to solve a **DCOP** with a given **DCOP** algorithm. These computations communicate with each other through message sending and a computation can only send messages to computations that it knows and depends on, forming a graph whose edges can the defined as a set $E_{\mathcal{C}}$. We note $\mathbf{N}(c_i) = \{c_j | (i,j) \in E_{\mathcal{C}}\}$ the set of neighbors of c_i in this graph.

Definition 5 (Computation Graph)**.** *A computation graph $G_{\mathcal{C}}$ is a tuple $\langle \mathcal{C}, E_{\mathcal{C}} \rangle$ where $\mathcal{C}$ is a set of*

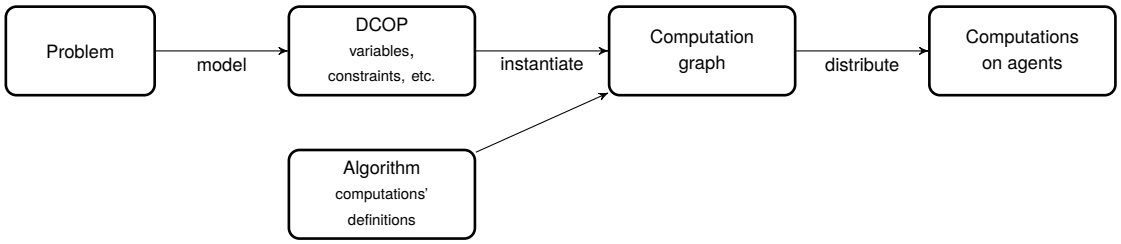


Figure 4.7 – Distribution of computations for an instantiated DCOP

computations and $E_\mathcal{C}$ is a set of edges (i, j) representing the dependencies between computations.

Of course, the exact set of computations, and their dependencies, needed for a given system depends on the **DCOP** used to model the problem (number of variables, constraints, etc.) and the **DCOP** algorithm used to solve that problem. Some $c \in \mathcal{C}$ are associated with a variable, whose value they are responsible for, while some other may serve other purposes. A **DCOP** algorithm defines what a computation does: what type of messages it reacts to, the structure and the content of the messages it sends, the conditions on which it selects a value for a variable (if applicable), etc. We can see a **DCOP** algorithm as a set of *computation definitions* and it's only when an algorithm is applied to a problem modeled as a **DCOP** that we have a set of computations, as depicted on Figure 4.7. As a matter of fact, the same **DCOP** can be instantiated as different sets of computations if solved with different algorithms.

Once these computations are defined, based on the **DCOP** and the algorithm, one must assign them to agents that will run them. We call this assignment a *distribution*.

Definition 6 (Distribution)**.** *Given a set of agents $\mathcal{A}$ and a computation graph $G_\mathcal{C} = \langle \mathcal{C}, E_\mathcal{C} \rangle$, a distribution is a mapping function $\nu : \mathcal{C} \mapsto \mathcal{A}$ that assigns each computation to exactly one agent.*

We note $a_a = \nu(c_i)$ the agent hosting the computation c_i and $\nu^{-1}(a_a)$ the set of computations hosted on agent a_a. Notice that this distribution ν is not necessarily the same function as the μ mapping function; while μ maps *variables* to agents, ν maps *computations* to agents.

- Many **DCOP** algorithms define exactly one computation for each variable, in which case the computation graph is equivalent to the constraint graph and $\mu = \nu$.
- However some algorithms define computations for each variables and constraints in the **DCOP**, and the distribution ν must account for the constraint computations, which are not considered by the mapping μ. When distributing computations for such algorithms, the computations form a factor graph $G_F = \langle \mathcal{C}_\mathcal{X}, \mathcal{C}_\mathcal{F}, E_F \rangle$ (see Section 2.3.1.1) where $\mathcal{C}_\mathcal{X}$ are the variable-bound computations, $\mathcal{C}_\mathcal{F}$ are the constraint-bound computations and E_F are the edges between these two types of vertices.
- Finally other algorithms, like for example partially centralized algorithms, may define a single computation representing several variables and/or constraints.

As we can see, the distribution is a more general concept than the mapping of variables to agents and better takes into account distributed implementation constraints. However, when such a mapping is given –even partially– by the problem we want to solve, it must obviously be respected. When defining a distribution, we should only distribute computations that are representing an element

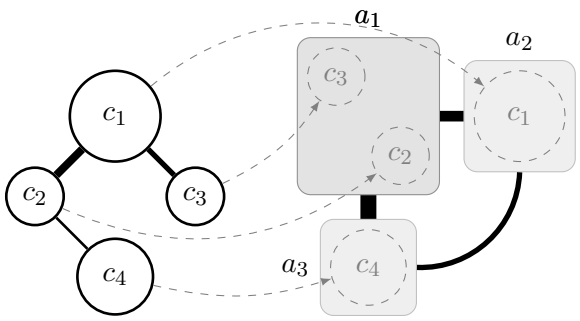


Figure 4.8 – A sample computation distribution problem

(variable or factor) that is not already mapped to an agent by μ.

For example, in the problem depicted in Figure 4.6a for the sensor network coordination problem, each variable is already mapped to an agent and the computations responsible for these variables must obviously respect this mapping. We only need to define a distribution for the factors F_{12} and F_{23}.

Formally, we have seen that on some real life problems, μ is often not defined for all $x \in \mathcal{X}$. We denote $\mathcal{X}_p$ the subset of $\mathcal{X}$ for which we have a mapping:

$$\mu : \mathcal{X}_p \mapsto \mathcal{A}, \quad \mathcal{X}_p \subseteq \mathcal{X} \tag{4.1}$$

As a computation represents one or several variables or constraints, the set of computations can be defined as follows, where ${}^{+}\mathbb{P}$ denotes the power set excluding the empty set:

$$\mathcal{C} \subset \mathbb{P}^{+}(\mathcal{X}) \cup \mathbb{P}^{+}(\mathcal{F}) \tag{4.2}$$

And in order to honor the mapping given by μ, $\nu : \mathcal{C} \mapsto \mathcal{A}$ must respect the following:

$$\forall c \in \mathcal{C}, \nu(c) = \begin{cases} \mu(c) & \text{if } c \in \mathcal{X}_p \\ a, a \in \mathcal{A} & \text{otherwise} \end{cases} \tag{4.3}$$

Now that we have defined the concept of distribution in a **DCOP**, we can explain why this approach is better suited to real world problems than the classical mapping of agents to variables.

4.2.2 Devising a Distribution

As seen previously, devising a distribution amounts to allocating any computation that is not already assigned by the mapping μ to an agent.

When devising this allocation, there are many elements that we can take into considerations. As a matter of fact, the placement of the computations on agents can have an important impact on the performance characteristics of the global system. Some distribution may improve response time, some other may favor communication load between agents and some others may be better for other criteria like QoS or running cost.

While distribution is seldom studied in the **DCOP** community, some recent works have started tackling it. For example, in [62] Khan et al. analyze the placement of constraint graph nodes on agents from a performance point of view; their objective is to find a placement that minimizes the completion time of the **DCOP**. Like us, they argue that in many problems there are multiple possible mappings of nodes to agents. However, they only consider variable nodes and do not define a more general concept of distribution, which takes into account other types of nodes (factors, several variables, etc.).

In in our work, we focus on applying **DCOP** for **AmI** scenarios. As stated in Section 3.2.1.1, the constrained resources in this kind of environment are the processing power of the connected devices, which act as agents, and the communication mechanism used among them, which is typically wireless, low power and has limited throughput. Consequently, our distribution mechanisms focus on these criteria and not specifically on the performance impact of the distribution, although we also experimentally assess that the run-time performance is acceptable.

The distribution problem can be seen as an instance of the graph partitioning problem, which typically falls under the category of NP-Hard problems [11, 34]. In graph partitioning, the vertices of a graph are assigned to mutually exclusive groups. Vertices and edges of the graph are commonly weighted, and the goal is to find a partition that minimize or maximize an objective function based on these weights. The most common objective is to minimize the edge cut, defined as the sum of the weights of the edges that cross between the groups. Devising a distribution also requires partitioning the set of computations into mutually exclusive groups, which maps the agents that run these computations, and the property expected from a distribution can be expressed using weights on elements of the computation graph.

As we can see, many different criteria can be used when devising a distribution for a **DCOP** and the definition of an optimal distribution is problem-dependent. Formally, once the criteria for optimality have been defined, finding an optimal distribution is an optimization problem in itself

In the following sections, we present several approaches for solving this problem and distributing **DCOP** computations in **AmI** environments.

4.3 A Naive Distribution for SECP

The **Smart Environment Configuration Problem (SECP)**, introduced in Section 3.2, is a **DCOP**-based model for coordination in **AmI** and **SHE**. In this model,

- the variables are either actuator variables $x_i \in \mathcal{X}(\mathfrak{A})$ or scene action variables $y_j \in \mathcal{X}(\Phi)$,
- the constraints are the energy cost functions of actuators $e_i \in \mathcal{F}(\mathfrak{A})$, the physical constraints $y_j \in \mathcal{X}(\Phi)$ and rule utility constraints $r_k \in \mathcal{F}(\mathfrak{R})$.

In the **SECP** model, we assume that each actuator node has a computation capability. We consider variables $x_i \in \mathcal{X}(\mathfrak{A})$ related to each actuator to be *owned* by their actuator's node, meaning they will always be deployed on this specific node. Therefore, a conventional μ mapping would associate actuator variables to their corresponding agent/device.

Our first distribution mechanism, introduced in [128], is a very simple heuristic which aims at reducing the communication load between agents by placing as much as possible computations

that communicate with one another one the same agent.

4.3.1 Distributing a Constraint Graph for SECP

When using an algorithm based on a constraint graph that only defines computations for variables, the computation graph is equivalent to the constraint graph. A conventional μ mapping would associate actuator variables to their corresponding agent/device and we only need to distribute the computations for the scene action variables y_j's.

In order to minimize communication load, we assign the computations for each y_j to one of the agents already hosting a variable that shares a constraint with y_j. This of course only holds if the algorithm supports n-ary constraints and we do not need to apply any binarization methods, which would introduce additional auxiliary variables that must be distributed on agents.

Given this assumption, our heuristic distribution for a constraint graph-based **SECP** can be defined as follows:

$$\begin{aligned} \mu : \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi) &\rightarrow \mathcal{A} \\ x_i &\mapsto a_i && \forall x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j &\mapsto a_i, x_i \in S_{\varphi_j} && \forall y_j \in \mathcal{X}(\Phi) \end{aligned} \tag{4.4}$$

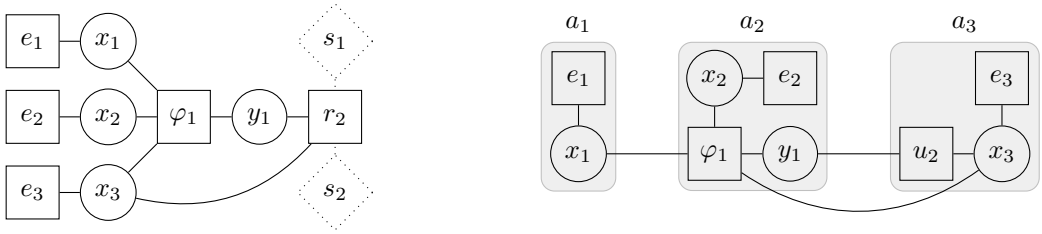
Definition 7 (GH-SECP-CGDP)**.** *We term **Greedy Heuristic for SECP Constraint Graph Distribution (GH-SECP-CGDP)** the method for distributing a constraint graph representing a **SECP** using that greedy heuristic.*

4.3.2 Distributing a Factor Graph for SECP

Here, we represent the **SECP** model using a factor graph as exposed in Section 3.2.2, in order to solve it using an algorithm like Max-Sum, which defines computations both for variables and constraints. We can safely argue that energy-cost functions, which are unary constraints, must be assigned to the same agent than the variables they are linked to. Consequently, we need to distribute the scene action variables, the physical constraints, which we consider as a pair $\langle y_j, \varphi_j \rangle$ and the rule utility constraints r_k.

To minimize the communication load, we place each pair $\langle y_j, \varphi_j \rangle$ on an agent a_i with i chosen such that $x_i \in S_{\varphi_j}$, meaning that x_i is one of the variables influencing y_j. Similarly, a factor r_k is hosted on an agent a_i such that $x_i \in S_{r_k}$. Intuitively this means that the factor representing a rule is always hosted on a agent affected by this rule. As to ensure a balanced computation load, y_j's, φ_j's and r_k's are fairly distributed among the candidate agents. This gives us the following definition for the distribution:

$$\begin{aligned} \nu : \mathcal{X}(\mathfrak{A}) \cup \mathcal{F}(\mathfrak{A}) \cup \mathcal{X}(\Phi) \cup \mathcal{F}(\mathbf{\Phi}) \cup \mathcal{F}(\mathfrak{R}) &\rightarrow \mathcal{A} \\ x_i &\mapsto a_i && \forall x_i \in \mathcal{X}(\mathfrak{A}) \\ e_i &\mapsto a_i && \forall e_i \in \mathcal{F}(\mathfrak{A}) \\ y_j &\mapsto a_i, x_i \in S_{\varphi_j} && \forall y_j \in \mathcal{X}(\Phi) \\ \varphi_j &\mapsto a_i, x_i \in S_{\varphi_j} && \forall \varphi_j \in \mathcal{F}(\mathbf{\Phi}) \\ r_k &\mapsto a_i, x_i \in S_{r_k} && \forall r_k \in \mathcal{F}(\mathfrak{R}) \end{aligned} \tag{4.5}$$



(a) Factor graph

(b) Sample distribution

Figure 4.9 – Factor graph and a possible distribution on 3 agents for a sample SECP

Figure 4.9 shows a sample distribution obtained using this heuristic.

Definition 8 (GH-SECP-FGDP)**.** *We term **Greedy Heuristic for SECP Factor Graph Distribution (GH-SECP-FGDP)** the method for distributing a factor graph representing a **SECP** using that greedy heuristic.*

Notice that these two definitions, for constraint graph and factor graph, still do not give us a fully defined distribution. Several agent assignments are typically valid for the variables y_j or the pairs $\langle y_j, \varphi_j \rangle$ and the factors r_k. When implementing this heuristic, we use a greedy approach to select one agent among the set of valid agents for a given computation: we select the agent, with enough capacity, that is already hosting the highest number of computations that share a dependency with the computation we are placing. In case of tie, we chose the agent with the highest remaining capacity. By grouping interdependent computations, this approach favors distributions with a low communication cost.

Of course, **GH-SECP-CGDP** and **GH-SECP-FGDP** are sub-optimal and offers no guarantee on the quality of the resulting distribution. We will see in Section 4.6.1 how they perform.

4.4 Optimal Distribution for SECP

As discussed previously, computing a distribution is equivalent to graph partitioning (see Section 4.2.2). While the approach discussed in the previous section is a naive heuristic that offers no guarantee on the quality of the distribution, we now model this problem as a mathematical optimization problem, which can be solved optimally. To scale up, we propose here an **Integer Linear Program (ILP)**, inspired by graph partitioning techniques from [16, 34].

In order to optimize the distribution for communication load, we need to formally define a measure of this load. We note $\mathbf{com}(c_i, c_j)$ the communication load induced by the interaction between c_i and c_j, which can be seen as the size of a message between these two computations and depends on the algorithm used. With a computation graph $G_\mathcal{C} = \langle \mathcal{C}, E_G \rangle$, we define the communication load as follows:

$$\forall c_i, c_j \in \mathcal{C} \quad \mathbf{com}(c_i, c_j) = \begin{cases} \text{message size}, & \text{if } (c_i, c_j) \in E_G \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

As the target environment for **SECP** is made of constrained devices, we also define $\mathbf{w_{max}}(a_k)$ the memory capacity in bytes of agent $a_k \in \mathcal{A}$. Let $\mathbf{mem}(c_i), c_i \in \mathcal{C}$ be the memory footprint for the

computation c_i. This can be, when using Max-Sum for instance, the size in bytes of the accumulated messages costs messages from neighbor variables, in a factor computation. representing the costs in a factor computation. When distributing computations on agents, we want to guarantee that the agents' capacities are not exceeded.

4.4.1 Distributing a Constraint Graph for SECP

We now devise an **ILP** for distributing a constraint graph-based computation graph $G_{\mathcal{C}} = \langle \mathcal{C}, E_{\mathcal{C}} \rangle$. First, let's introduce a set of binary variables that map computations to agents: c_i^k denotes whether variable c_i is distributed on agent a_k:

$$\forall c_i \in \mathcal{C}, \quad c_i^k = \begin{cases} 1, & \text{if } \nu(c_i) = a_k \\ 0, & \text{otherwise} \end{cases}$$

Additionally we assume that the communication load between computations run by the same agent is negligible, as it will use some kind of local communication mechanism (shared memory, anonymous pipe, etc.) which is typically orders of magnitude faster that network communication. We introduce another set of variables α_{ijk} which indicates if two computations are distributed on the same agent:

$$\forall c_i, c_j \in \mathcal{C}, a_k \in \mathcal{A}, \quad \alpha_{ijk} = c_i^k \cdot c_j^k$$

Using these definitions, we can express the total communication load as follows,

$$\sum_{(c_i,c_j)\in E_{\mathcal{C}}} \sum_{a_k \in \mathcal{A}} \mathbf{com}(c_i, c_j) \cdot (1 - \alpha_{ijk})$$

This definition of α_{ijk} leads to a quadratic formulation but can be linearized as proposed in [3, 16]:

$$\alpha_{ijk} \leq c_i^k, \quad \alpha_{ijk} \geq c_i^k + c_j^k - 1$$

With a slight abuse of notation, we use $\mathcal{X}(\mathfrak{A}) \subset \mathcal{C}$ to denote the set of computations representing actuator variables. These computations are naturally mapped to their corresponding agent:

$$\forall c_i \in \mathcal{X}(\mathfrak{A}), \quad \exists a_k \in \mathcal{A}, \quad c_i \in \mu^{-1}(a_k)$$

This can be expressed by the following constraint:

$$\forall a_k \in \mathcal{A}, \forall c_i \in \mu^{-1}(a_k), \quad c_i^k = 1$$

Finally, to account for devices with limited memory, we add a constraint to avoid memory capacity overflow:

$$\forall a_k \in \mathcal{A}, \quad \textstyle\sum_{c_i \in \mathcal{C}} \mathbf{mem}(c_i) \cdot c_i^k \leq \mathbf{w_{max}}(a_k)$$

Now, we are ready to model the distribution of the computation graph used to solve our **SECP** as

a linear program:

Definition 9. *(ILP for constraint graph SECP)*

$$\underset{c_i^k, c_j^k}{\textbf{\textit{minimize}}} \quad \sum_{(c_i,c_j)\in E_\mathcal{C}} \sum_{a_k \in \mathcal{A}} \mathbf{com}(c_i, c_j) \cdot (1 - \alpha_{ijk}) \tag{4.7}$$

subject to

$$\forall c_i \in \mathcal{C}, \quad \sum_{a_k \in \mathcal{A}} c_i^k = 1 \tag{4.8}$$

$$\forall a_k \in \mathcal{A}, \quad \sum_{c_i \in \mathcal{C}} c_i^k \geq 1 \tag{4.9}$$

$$\forall a_k \in \mathcal{A},\ \forall c_i \in \mu^{-1}(a_k), \quad c_i^k = 1 \tag{4.10}$$

$$\forall a_k \in \mathcal{A}, \quad \textstyle\sum_{c_i \in \mathcal{C}} \mathbf{mem}(c_i) \cdot c_i^k \leq \mathbf{w_{max}}(a_k) \tag{4.11}$$

$$\forall (c_i, c_j) \in E_\mathcal{C}, \quad \alpha_{ijk} \leq c_i^k \tag{4.12}$$

$$\forall (c_i, c_j) \in E_\mathcal{C}, \quad \alpha_{ijk} \geq c_i^k + c_j^k - 1 \tag{4.13}$$

Objective (4.7) minimizes communications between computations which are not distributed on the same agent. Constraint (4.8) forces each computation from the computation graph to be deployed on exactly one agent. Constraint (4.9) enforces the use of all the available agents. Constraint (4.10) enforces the placement of actuator variables on the agents representing these devices. Finally constraints (4.12) and (4.13) link c_i^k's and c_j^k's to α_{ijk} in a linear way.

Definition 10 (ILP-SECP-CGDP)**.** *We term **Integer Linear Program for SECP Constraint Graph Distribution (ILP-SECP-CGDP)** the 0/1 integer linear program consisting of objective (4.7) and constraints (4.8) to (4.13) which encodes the problem of distributing a constraint graph representing a **SECP**.*

Example 11. *We consider a **SECP**, presented by Figure 4.10a, with 3 actuators $\{x_1, x_2, x_3\}$, two physical model $\{\phi_1, \phi_2\}$ and one rule r_1.*

*When using DSA, or any other constraint graph based algorithm, the **DCOP** for this **SECP** results in a computation graph with 5 computations, depicted on Figure 4.10b.*

As actuator variables are already mapped to devices, we only need to distribute the computations $c_{y_1} and c_{y_2}$ corresponding to physical model variables.

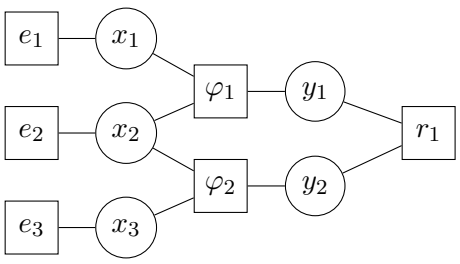
When using DSA, messages contain a single value:

$$\forall c_i, c_j, \quad \mathbf{com}_{DSA}(c_i, c_j) = 1$$

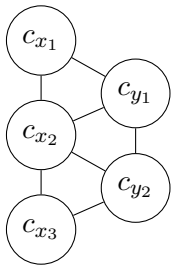
The footprint of a computation is proportional to the number of neighbors in the computation graph:

$$\forall c_i, \quad \mathbf{mem}_{DSA}(c_i) = |\mathbf{N}(c_i)|$$

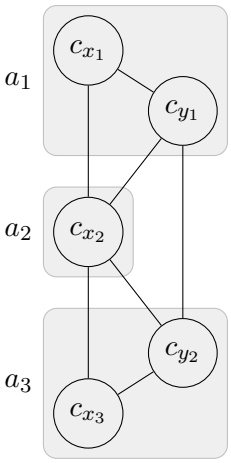
If all agents have a capacity $\mathbf{w_{max}}(c_i) = 5$, when applying the ILP (Definition 9), computations c_{y_1} and c_{y_2} are distributed on agents a_1 and a_3, as depicted on Figure 4.10c to respect capacity



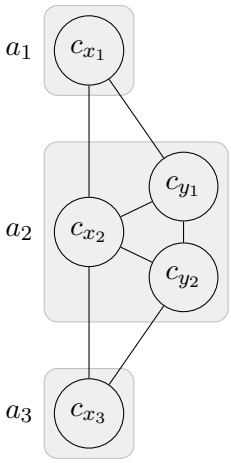
(a) Factor graph representation



(b) Computation graph



(c) When all capacities are equal to 5



(d) When a_2 capacity is equal to 10

Figure 4.10 – **ILP** based distributions for a simple **SECP** with 3 actuators

constraints.

On the other hand, if agent a_2 has a capacity of 10*, these computations are distributed on agent a_2, as represented on Figure 4.10d which reduces communication load overall.*

We will see in 4.4.3 how and when this **ILP** can be used to distribute a **DCOP** representing a **SECP**.

4.4.2 Distributing a Factor Graph for SECP

When our **SECP** is modeled as a factor graph $G_F = \langle \mathcal{C}_\mathcal{X}, \mathcal{C}_\mathcal{F}, E_F \rangle$, an interaction can only occur between a variable computation and a factor computation.

In that case, $\mathbf{com}$ is the size of the messages exchanged between the computation $c_i \in \mathcal{C}_\mathcal{X}$ representing the variable x_i and the computation $c_j \in \mathcal{C}_\mathcal{F}$ representing the factor f_j. For easier reading, and with a slight abuse of notation, we denote $x_i \in \mathcal{C}_\mathcal{X}$ the computation c_i representing variable x_i and $f_j \in \mathcal{C}_\mathcal{F}$ the computation c_j representing factor f_j:

$$\forall x_i \in \mathcal{C}_\mathcal{X}, f_j \in \mathcal{C}_\mathcal{F}, \quad \mathbf{com}(x_i, f_j) = \begin{cases} \text{message size}, & \text{if } (x_i, f_j) \in E_F \\ 0, & \text{otherwise} \end{cases}$$

Similarly to the constraint graph version, we introduce a set of binary variables that map factor graph elements to agents: x_i^k (respectively f_j^k) denotes whether variable x_i (respectively factor f_j)

is distributed on agent a_k and α_{ijk}:

$$\forall x_i \in \mathcal{C}_\mathcal{X}, \quad x_i^k = \begin{cases} 1, & \text{if } \nu(x_i) = a_k \\ 0, & \text{otherwise} \end{cases}$$

$$\forall f_j \in \mathcal{C}_\mathcal{F}, \quad f_j^k = \begin{cases} 1, & \text{if } \nu(f_j) = a_k \\ 0, & \text{otherwise} \end{cases}$$

$$\forall x_i \in \mathcal{C}_\mathcal{X}, f_j \in \mathcal{C}_\mathcal{F}, a_k \in \mathcal{A}, \quad \alpha_{ijk} = x_i^k \cdot f_j^k$$

As previously, we also add constraints ensuring that computations for actuator variable and energy-cost factors are placed on the agent they are bound to by the μ mapping:

$$\forall a_k \in \mathcal{A},\ \forall c_i \in \mu^{-1}(a_k), \quad c_i^k = 1$$

Finally, as SECP deals with devices with limited memory, we add a constraint to avoid memory capacity overflow:

$$\forall a_k \in \mathcal{A}, \quad \textstyle\sum_{c_i \in \mathcal{C}_\mathcal{X} \cup \mathcal{C}_\mathcal{F}} \mathbf{mem}(c_i) \cdot c_i^k + \ \leq \mathbf{w_{max}}(a_k)$$

Using these notations, we model the distribution of the factor graph representing our **SECP** as a linear program:

Definition 11. *(ILP for factor graph distribution)*

$$\underset{x_i^k, f_j^k}{\textbf{\textit{minimize}}} \quad \sum_{(x_i,f_j)\in E_F} \sum_{a_k \in \mathcal{A}} \mathbf{com}(x_i, f_j) \cdot (1 - \alpha_{ijk}) \tag{4.14}$$

subject to

$$\forall x_i \in \mathcal{C}_\mathcal{X}, \quad \sum_{a_k \in \mathcal{A}} x_i^k = 1 \tag{4.15}$$

$$\forall f_j \in \mathcal{C}_\mathcal{F}, \quad \sum_{a_k \in \mathcal{A}} f_j^k = 1 \tag{4.16}$$

$$\forall a_k \in \mathcal{A}, \quad \sum_{x_i \in \mathcal{C}_\mathcal{X}} x_i^k \ + \sum_{f_j \in \mathcal{C}_\mathcal{F}} f_j^k \geq 1 \tag{4.17}$$

$$\forall a_k \in \mathcal{A},\ \forall c_i \in \mu^{-1}(a_k), \quad c_i^k = 1 \tag{4.18}$$

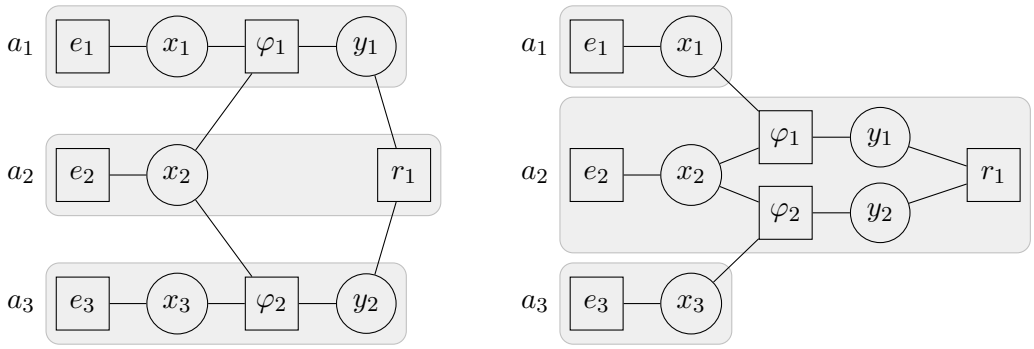
$$\forall a_k \in \mathcal{A}, \quad \textstyle\sum_{c_i \in \mathcal{C}_\mathcal{X} \cup \mathcal{C}_\mathcal{F}} \mathbf{mem}(c_i) \cdot c_i^k + \ \leq \mathbf{w_{max}}(a_k) \tag{4.19}$$

$$\forall (x_i, f_j) \in E_f, \quad \alpha_{ijk} \leq x_i^k \tag{4.20}$$

$$\forall (x_i, f_j) \in E_f, \quad \alpha_{ijk} \leq f_j^k \tag{4.21}$$

$$\forall (x_i, f_j) \in E_f, \quad \alpha_{ijk} \geq x_i^k + f_j^k - 1 \tag{4.22}$$

Definition 12 (ILP-SECP-FGDP)**.** *We term **Integer Linear Program for SECP Factor Graph Distribution (ILP-SECP-FGDP)** the 0/1 integer linear program consisting of objective (4.14) and constraints (4.15) to (4.22) which encodes the problem of distributing a factor graph representing a **SECP**.*



(a) When agents have limited capacity

(b) When capacity is not an issue

Figure 4.11 – **ILP** based distributions for the **SECP** from Figure 4.10

As previously, objective (4.14) minimizes communications between computations which are not distributed on the same agent. Other constraints ensure that each computation is distributed on exactly one agent (4.16, 4.15), that every agent is hosting at least one computation (4.17) while respecting agents' capacities (4.19) and respecting the μ mapping (4.18). Finally (4.20), (4.21) and (4.22) linearize the program.

Example 12. *When using a factor graph based algorithm, like Max-Sum, to solve the SECP from Example 11, the computation graph maps the factor graph (depicted on Figure 4.10a).*

We now need to distribute computations for the factor nodes (actuators' energy costs, physical models and rule): $\{e_1, e_2, e_3, \varphi_1, \varphi_2, r_1\}$ *and the physical model variable nodes:* $\{y_1, y_2\}$.

When using Max-Sum, the message size is a function of the size of domain of the variable:

$$\mathbf{com}_{MaxSum}(x_i, f_j) = \mathbf{com}_{MaxSum}(f_j, x_i) = |\mathcal{D}_{x_i}|$$

A variable computation stores, for each of it's neighbors, the best known cost and the associated value; thus it's footprint is proportional to the size of the variable's domain and the number of neighbors. Factor computations need to store the last message from each of their neighbors, and thus their footprint is a function of the sum of their neighbors' domain size.

$$\begin{aligned} \forall x_i, \quad \mathbf{mem}_{MaxSum}(x_i) &= |\mathbf{N}(x_i)| \cdot \mathcal{D}_{x_i} \\ \forall f_j, \quad \mathbf{mem}_{MaxSum}(f_j) &= \sum_{x_i \in \mathbf{N}(f_j)} \mathcal{D}_{x_i} \end{aligned}$$

When agents have limited capacity (for example 40*, which the minimum needed in that case to make distribution possible), applying ILP (Definition 11) results in the distribution represented on Figure 4.11a.*

On the other hand, if capacity is not an issue (for example when assigning a capacity of 100 *to each agent) the ILP distributes all 'distributable' computations (i.e. distribution that do not have a mapping) on a single agent in order to minimize communication load, as represented on Figure 4.11b.*

4.4.3 Solving the ILP for SECP Distribution

We have devised two models for distributing a computation graph representing a **SECP**. Our distribution problem has been modeled as an **ILP**, and of course is NP-hard. To solve this **ILP**, we can use efficient off-the-shelf solvers, commercial or open source, like Gurobi [1], CPLEX [2] or GLPK [3].

When the instance is not too large it can be solved in reasonable time in a centralized manner with branch-and-cut algorithm [85]. This algorithm is particularly efficient when the coefficient matrix is sparse. which is generally the case with a **SECP**, because the constraint graph of such problems is generally sparse and have some dense subgraphs corresponding to rooms and rules. Even though these solvers are very efficient, they solve the problem in a centralized manner, which is far from ideal in our target **AmI** and **IoT** environments. When boostrapping our system, this might not be an issue: we can easily assume that the first time the system is installed in the home environment, some kind of central computing device is available that can be used to compute an optimal initial distribution for all the computations of our **DCOP**-modeled **SECP**.

However, when modeling the distribution problem for our **SECP** as an **ILP**, the resulting **ILP** is a model of the full problem, including all devices and all the user defined rules, which are dynamic by nature. This means that whenever a user add or remove a rule to the system, or when the definition of an existing rule is modified, we should recompute the distribution. We could consider that at these moments, as the user is modifying the set of rules, there might be some device available where we could run the solver. For example the user interface may be running on a tablet or another kind of mobile device that is powerful enough for this task. This, however, unfortunately rules out any lightweight user-interface approach, which should be the target in the **AmI** philosophy, where the user interface should be non-intrusive and even blends into the environment itself (see Section 2.1.1).

Even if we can recompute the distribution using a centralized solver when the set of rules is modified, the situation is different when devices enter or leave the system. In such events, the distribution should also the re-computed, yet we do not have the possibility to solve the **ILP** on a off-the-shelf solver, as the only available devices are the actuators and sensors, which are typically low power devices. For these reasons, it would be ideal to be able to compute a distribution, or at least fix an existing one, in a distributed manner.

There are some works that tackle the problem of solving **ILP** in a distributed way; for example in [19] Burger et al. introduce a distributed simple algorithm designed specifically for multi-agent tasks assignment, which is very close to our own problem. However, even in this distributed version, computing the solution of the **ILP** on a computationally limited device is not realistic. We will see in Section 5 how we can deal with such dynamics in our **AmI**, and in general **IoT** settings.

It should be noted that even when the **ILP** representation of the distribution problem cannot be solved, either because it is too big or because there is no device available to run the solver, it is still useful as its objective function provides us with a quality metrics that can be used to evaluate

1. http://www.gurobi.com/
2. https://www.ibm.com/analytics/cplex-optimizer
3. https://www.gnu.org/software/glpk

comparatively several non-optimal distributions that have been obtained through heuristics or approximate algorithms. We use this measure of quality for our experimental evaluations.

4.5 A Generalized Definition of Optimal Distribution for IoT Systems

In previous sections, we devised models for distributing a computation graph representing a **DCOP** used to solve the **SECP** introduced in Section 3.1. Here, we extend this model to a more generic definition of optimal distribution of computations in **IoT** systems.

4.5.1 Computation Graph Models

The concept of organizing many computations as a graph, with edges representing the dependencies and communication between computations, is not specific to **SECP** nor **DCOPs**; many other computation models use this kind of organisation.

Computation graphs were initially introduced by Karp and Miller in [61] for modelling parallel computing systems. In distributed computing, they are also generally known as *Dataflow models* and have many variants like Kahn Process Network [59], or Synchronous Dataflow Model [72], to name just a few.

The **Bulk Synchronous parallel (BSP)** model, introduced by Valiant in [141], and frameworks implementing it like Pregel [80] or Apache Hama [4] for instance, leverage distributed computing to process big data. In this model, the goal is to parallelize computations as much as possible, in order to speed up the overall completion time by running then simultaneously on several computers.

Another example where computation graph are used in practice is **Network Function Virtualization (NFV)**, currently studied to simplify network management. **Virtual Network Function (VNF)**, which are essentially computations operating on network packets, are organized as graphs called **VNF Forwarding Graph (VNF-FG)** and must be allocated to a physical infrastructure, in a way that minimizes the utilisation of physical infrastructure while meeting the QoS requirements, as discussed by Moens et al. in [87].

Recently, computation graphs have also been used to describe the computations required for deep learning models and several works study the efficient distribution of the nodes of these graphs to speed up computation [1, 150]. In all these models, the nodes of the graph represent tasks, or computations, that must be placed on a physically infrastructure to run, and thus they are all, at least to some extend, concerned with the distribution problem.

The placement of these computations has an important impact on the performance characteristics of the global system: for example some distributions may improve response time by allowing parallelism, some other may favor the communication load between the nodes and some other may be better for other criteria like QoS, running and hosting cost, etc.

As discussed in 4.2.2, the distribution problem is equivalent to graph partitioning and the definition of the objective depends on the problem domain. In our case, as we target **AmI** and more generally **IoT** systems, our model takes two objectives into account for the definition of an optimal distribution: communication efficiency and agents' cost to host computations. Additionally, the

4. http://hama.apache.org/

distribution must satisfies physical constraints on the agent's capacity.

4.5.2 Problem Definition

While the models previously proposed in Sections 4.4.1 and 4.4.2 for the **SECP** were designed to compute an optimal distribution for a computation graph representing a **DCOP**, the model we introduce here targets generic computation graphs and add several features that we consider to be necessary in **IoT** systems.

Let $G_{\mathcal{C}} = \langle \mathcal{C}, E_{\mathcal{C}} \rangle$ be a computations graph, and let $\mathcal{A}$ be the set of agents which can host the computations $c_i \in \mathcal{C}$.

Communications. In **IoT** settings, communications are heterogeneous both in performance characteristics and costs; they can range from hight speed fiber connection to cloud servers to low-power, short-range, slow wireless connection between constrained devices in the Local Area Network. As a consequence, for a distribution to be communication-efficient it should both generate as little communication load as possible and favor the cheapest communication links.

We assume that all agents can potentially communicate with each other and model the communication cost with a cost matrix: $\mathbf{route}\colon \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}^+$ where $\mathbf{route}(a_m, a_n)$ is the communication cost between agents a_m and a_n.

We note $\mathbf{msg}(c_i, c_j)$ the size of the messages between the computations c_i and c_j. Using these functions, we can define the communication cost between the computation c_i hosted on agent a_m and c_j hosted on a_n as follows:

$$\forall c_i, c_j \in \mathcal{C}, \quad \forall a_m, a_n \in \mathcal{A}, \quad \mathbf{com_a}(c_i, c_j, a_m, a_n) = \mathbf{msg}(c_i, c_j) \cdot \mathbf{route}(a_m, a_n) \tag{4.23}$$

Notice that this model for communication cost is quite flexible. We can, as previously, decide that there is no communication cost when the two computations are hosted on the same agent by simply setting:

$$\forall a_m \in \mathcal{A},\ \mathbf{route}(a_m, a_m) = 0$$

We could also assign a (typically small) non-null cost for intra-agent communication and even specify different intra-agent communication costs depending on the type of agents.

This approach also allows modeling systems where some agents cannot communicate with some other agents, by simply assigning infinite costs in this cost matrix.

Hosting Computations. When hosting a computation in an **IoT** environment, it is often desirable to favor some agents for other reasons than communication only. For example some computation might be very CPU intensive and we want to ensure it will be hosted on a server with a powerful CPU. When using cloud resources, there might also be different cost for hosting a computation on a given server, compared to another. Additionally some parts of the infrastructure might be less prone to disconnection and some computations may be less tolerant to sporadic connection. Finally,

especially in **IoT**, some computation might be tightly linked to one specific physical element, as it is for example the case in our **SECP** model, and can only be reasonably hosted on that element.

We model this affinity, or repulsiveness, between an agent and a computation with a function $\mathbf{cost}\colon \mathcal{A} \times \mathcal{C} \mapsto \mathbb{R}^+$ that assign a cost for each pair (a_m, c_j).

Notice that one can easily force a computation $c_{\mathbf{j}}$ to be hosted on a specific agent $a_{\mathbf{n}}$ by assigning an infinite hosting cost for all other agents:

$$\forall c_i \in \mathcal{C}, \forall a_m \in \mathcal{A}, \quad \mathbf{cost}(c_i, a_m) = \begin{cases} 0 & \text{if } i = \mathbf{j} \text{ and } m = \mathbf{n} \\ \infty & \text{otherwise} \end{cases}$$

Capacity. As in previous models, we also consider that an agent can only host a limited number of computations and model this with agents' capacity and computations' footprint noted respectively $\mathbf{w_{max}}(a_m)$ and $\mathbf{mem}(c_i)$

Using these definitions, we define an **IoT** optimal distribution of a computation graph as follows:

Definition 13 (**IoT** optimal distribution)**.** *An **IoT optimal distribution** is a distribution ν that minimizes the cost of communication between agents and minimize the cost of hosting computations while respecting the agents' capacity constraints.*

Definition 14 (CGDP)**.** *Given a computation graph and a set of agents, the **Computation Graph Distribution Problem (CGDP)** amounts to assign each computation of the computation graph to an agent to obtain an **IoT** optimal distribution.*

4.5.3 Linear Program for Optimal Distribution

We can now encode our **IoT** optimal distribution problem as **ILP**, as we did for the **SECP**.

Let c_i^m be a binary variable denoting whether the computation c_i is hosted on agent a_m. The binary variable and α_{ij}^{mn} denotes if both computation c_i is hosted on agent a_m and c_j is hosted on a_n.

$$\forall c_i \in \mathcal{C}, \quad c_i^m = \begin{cases} 1, & \text{if } \nu(c_i) = a_m \\ 0, & \text{otherwise} \end{cases}$$

$$\forall\, c_i, c_j \in\ \mathcal{C}, \quad \alpha_{ij}^{mn} = c_i^m \cdot c_j^n$$

Our communication efficiency objective amounts to minimize the communication cost for all edges of the computations graph and can be written as follow:

$$\underset{c_i^m}{\textbf{minimize}} \quad \sum_{(i,j)\in E_{\mathcal{C}}} \quad \sum_{(m,n)\in \mathcal{A}^2} \mathbf{com_a}(c_i, c_j, a_m, a_n) \cdot \alpha_{ij}^{mn}$$

Additionally, the hosting cost objective can be written as follow:

$$\underset{c_i^m}{\textbf{minimize}} \sum_{(c_i, a_m)\in X \times \mathcal{A}} c_i^m \cdot \mathbf{cost}(a_m, c_i)$$

By aggregating these two objectives with penalizing factors $\omega_{\mathbf{com}}$ and $\omega_{\mathbf{c_{host}}}$, we can define our distribution problem as a mono-objective optimization problem, modeled through the **ILP**:

$$\begin{aligned}
\underset{c_i^m}{\textbf{minimize}} \quad & \omega_{\mathbf{com}} \cdot \sum_{(i,j)\in E_{\mathcal{C}}} \sum_{(m,n)\in\mathcal{A}^2} \mathbf{com_a}(c_i, c_j, a_m, a_n) \cdot \alpha_{ij}^{mn} \\
+ \quad & \omega_{\mathbf{c_{host}}} \cdot \sum_{(c_i,a_m)\in\mathcal{C}\times\mathcal{A}} c_i^m \cdot \mathbf{cost}(a_m, c_i) \qquad (4.24)
\end{aligned}$$

subject to

$$\forall a_m \in \mathcal{A}, \quad \sum_{c_i\in\mathcal{C}} \mathbf{mem}(c_i) \cdot c_i^m \leq \mathbf{w_{max}}(a_m) \qquad (4.25)$$

$$\forall c_i \in \mathcal{C}, \quad \sum_{a_m\in\mathcal{A}} c_i^m = 1 \qquad (4.26)$$

$$\forall c_i \in \mathcal{C}, \quad \alpha_{ij}^{mn} \leq c_i^m \qquad (4.27)$$

$$\forall c_j \in \mathcal{C}, \quad \alpha_{ij}^{mn} \leq c_j^m \qquad (4.28)$$

$$\forall c_i, c_j \in \mathcal{C}, a_m \in \mathcal{A}, \quad \alpha_{ij}^{mn} \geq c_i^m + c_j^n - 1 \qquad (4.29)$$

This **ILP** is flexible enough to accommodate a large panel of **IoT** scenarios; by using appropriate communication and hosting cost matrices one can for example easily use it to reproduce the **ILP**s designed specifically for a **DCOP** representing a **SECP**, both for constraint graph or a factor graph based algorithms.

Definition 15 (ILP-CGDP)**.** *We term **Integer Linear Program for CGDP (ILP-CGDP)** the 0/1 integer linear program consisting of objective (4.24) and constraints (4.25) to (4.29) which encodes the **CGDP** problem (Definition 14).*

This program gives us a definition of an optimal distribution, and can be solved by classical centralized solvers. However, the complexity is still very hard and it is only possible for relatively small systems. **AmI** systems, and more generally **IoT** systems can be very large, meaning that this approach would most probably not scale. Even though, the objective function can still be used to evaluate the quality of approximate distribution method, as we do when evaluating our approaches experimentally in Section 4.6.

4.5.4 Greedy Heuristic for Computation Graph Distribution

As **ILP-CGDP** might be too difficult to solve for large system, we also developed a heuristic that computes an approximate distribution. This approach is a greedy heuristic similar to those introduced in Sections 4.3.1 and 4.3.2, except communication costs now takes into account the route, as we do for **ILP-CGDP**. Moreover, this heuristic is designed for generic computation graph and thus works for both constraint graph and factor graph based algorithms.

We start by placing the computation with the highest footprint, and select the agent with enough remaining capacity that incurs the lowest aggregate communication and hosting costs. In case of ties, we chose the agent with the highest remaining capacity.

Of course, this greedy heuristic is suboptimal, but allows computing a distribution easily even for

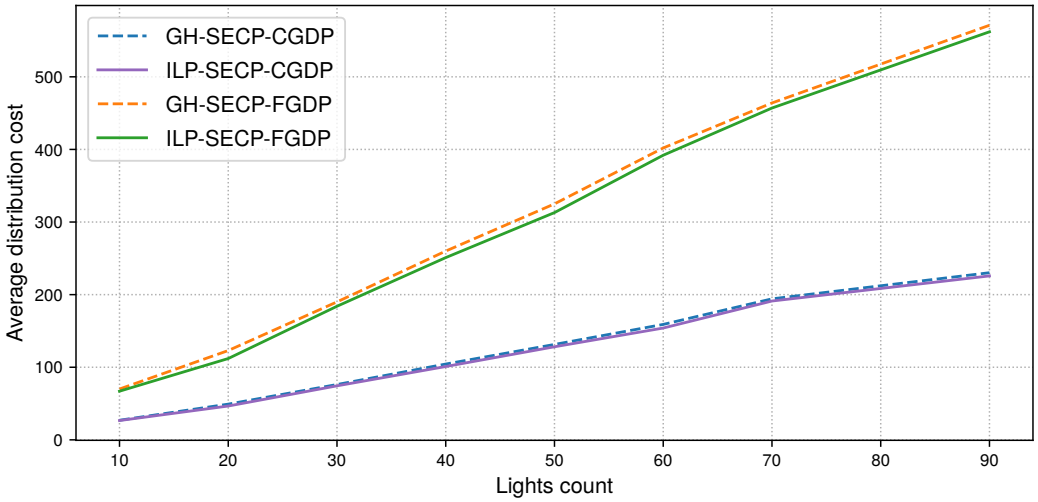


Figure 4.12 – Distribution costs for increasing size **SECP** instances with several **SECP**-specific distribution methods

large systems. Notice however that if the system is severely constrained capacity-wise, it may fail to find a distribution even though one exists and would be found by **ILP-CGDP** (provided the **ILP** can be solved in a reasonable time).

Definition 16 (GH-CGDP)**.** *We term Greedy Heuristic for CGDP (GH-CGDP) the method for distributing a computation graph using that greedy heuristic.*

4.6 Experimental Evaluations

In this section, we evaluate our distribution methods on several problems types, both on randomly generated **SECP** instances and on classical benchmark problems.

4.6.1 Evaluation of SECP-specific Distribution Methods

First, we evaluate our distribution methods for **SECP**: **GH-SECP-CGDP** and **ILP-SECP-CGDP** for constraint graph based algorithms and **GH-SECP-FGDP** and **ILP-SECP-FGDP** for factor graph based algorithms.

We generate random **SECP** instances with the same methods used in Section 3.3, with a growing number of lights, physical models and rules.

Then, we distribute each instance using our four solutions methods. We used **DSA** for the constraint graph algorithm and **MaxSum** for the factor graph algorithm.

Figure 4.12 shows the costs of the distribution obtained using these four methods. Notice that the distribution costs, for the same original problem, for a constraint graph and a factor graph algorithm should not be compared: the costs represents the communication load between nodes in the graph that are not placed on the same device, and a factor graph representation induces many more computations than a constraint graph.

We can see from this figure that our two heuristics, **GH-SECP-CGDP** and **GH-SECP-FGDP**

produce distribution whose cost is very similar to the optimal distribution from the **ILP** based methods **ILP-SECP-CGDP** and **ILP-SECP-FGDP**. The optimal distribution's costs are only 3% better that the approximate distribution produced by our heuristics.

Notice however that, while it never happened in our experiments, it is possible to have a **SECP** that the heuristics fail to distribute even though a distribution exists, and is of course found when using the **ILP** approach.

Figure 4.13 represents the average time, in seconds with a logarithmic scale, required to compute each of these distributions; we can clearly see that the heuristics are much faster than the optimal methods, especially when the problems grow larger. Our heuristics never need more than 0.1 seconds while the time grows exponentially when using an **ILP**.

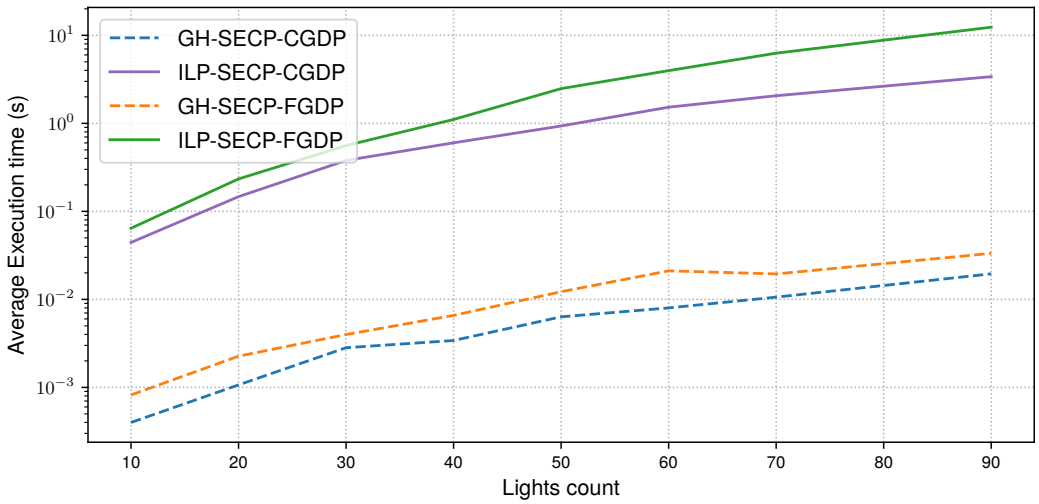


Figure 4.13 – Times for computing a distribution for increasing size **SECP** instances with several **SECP**-specific distribution methods

4.6.2 Evaluating the Generalized Distribution for IoT Systems on Benchmark Problems

In this first experiment we use our optimal and heuristic distribution methods on two different types of **DCOP**s traditionally used for benchmarks: random graph soft coloring problems and scale free graph coloring problems.

Random soft coloring problems are generated by creating a random graph with density $p = 0.2$. Each edge is then mapped to a binary constraint (whose cost function has no influence on the distribution) and each vertex is mapped to a variable with a domain made of 5 colors. Scale free problems are derived, using the same approach, from scale free graphs generated using the Barabàsi-Albert model [8] (starting from a 2-node connected graph), which are known to adequately model **IoT** systems [151]. In both cases, we generate instances with an increasing number of variables: $|\mathcal{X}| \in \{12, 24, \ldots, 84\}$. For each variables count, we generate 10 instances.

Then, for each problem instance, we generate three different infrastructures made of a set of agents, such that the number of variables is 2, 3 or 4 bigger than the number of agents (denoted 'da' for density agent in the figures' legends). This allows us to evaluate the effect of the density of the

system, when each agent must host on average more computations.

These problems are then mapped to two different computation graphs, for **DCOP** algorithms based on constraint graph and factor graphs. Finally, we use our two methods, **ILP-CGDP** and **GH-CGDP** to compute a distribution based on our concept of a generalized definition of computations distribution for **IoT** systems.

We use pyDCOP (see Section 6) for generating the problem instances, the computation graph and the agents, and for computing the distributions. For solving the linear program **ILP-CGDP** is based on, pyDCOP relies on the GLPK [5] solver and a time budget of 30 seconds is used.

Figure 4.14 represents the time, in seconds with a logarithmic scale, required to compute the distribution of random graph coloring problems when using an algorithm based on a constraint graph representation, with several agent densities, using the two methods. As we can see **ILP-CGDP** is very quickly blocked by the 30-seconds time limit, even though the constraint graph representation requires few computation compared to a factor graph representation. When removing that limit, it requires an unreasonable amount of time, up to more than 45 minutes, for a problem with 72 variables. Distribution becomes harder as the problems grow in size, and when the average number of computation per agent gets lower: we manage to compute a distribution for up to 48 variables when $da = 4$ but all distributions with more than 36 agents fail when $da < 4$. On the other hand the distribution time when using **GH-CGDP** stays very small and, while it also increases with the problem size, the number of computations for each agent has little effect.

When using a factor graph representation, the situation is similar but the problem is even harder and we can only distribute optimally problems with 24 variables using the **ILP** with a 30 seconds time limit, as depicted on Figure 4.15. The **GH-CGDP** heuristic manages to compute a distribution for all instances, but the time required increases very quickly. The difference with the results when using a constraint graph representation can be explained by the fact that a factor graph representation of the same problem requires many more computations, as we have computations for variables and constraints. For example, an instance with $|\mathcal{X}| = 82$ variables and a graph density of $p = 0.2$ has $\frac{82\times(82-1)\times0.2}{2} \approx 664$ edges, each of which represents one constraint. Thus, we have to distribute $664 + 82 = 746$ computations, while the constraint graph only requires to distribute 82 computations, one for each variable in the problem.

5. https://www.gnu.org/software/glpk/

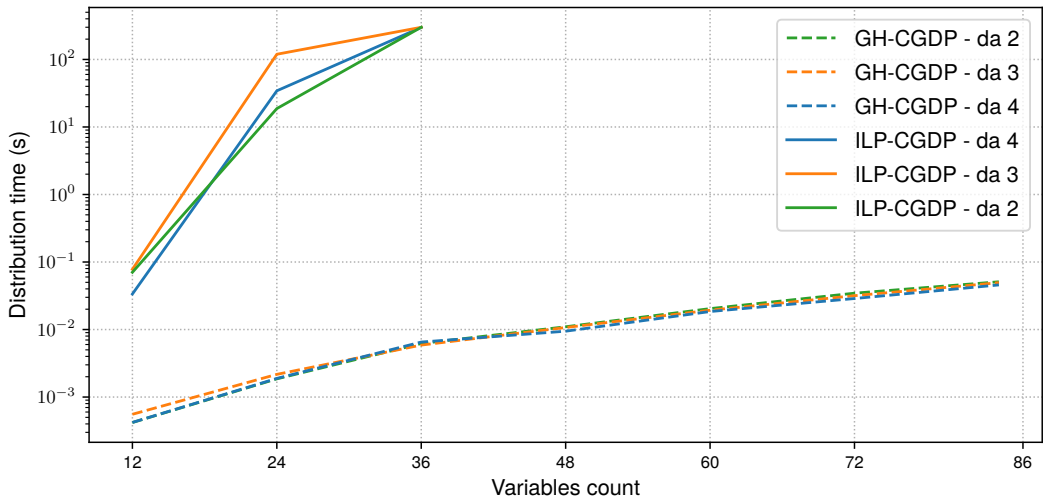


Figure 4.14 – Time for distributing random graph coloring problems with optimal and heuristic methods, when using a constraint graph representation

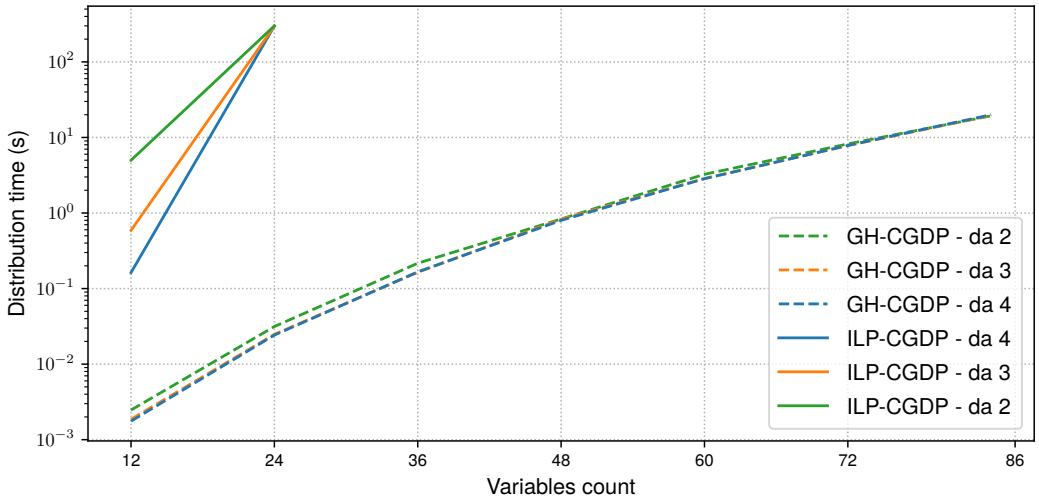


Figure 4.15 – Time for distributing random graph coloring problems with optimal and heuristic methods, when using a factor graph representation

Figure 4.16 and 4.17 represent the costs of the distributions produced by **ILP-CGDP** and **GH-CGDP** on random graph coloring problems, when using respectively a constraint graph and factor graph representation. As optimal distributions for large problems could not be computed using **ILP-CGDP**, only smaller instances are plotted. We can see that the suboptimal distribution methods produces very good quality results, while requiring several orders of magnitude less time.

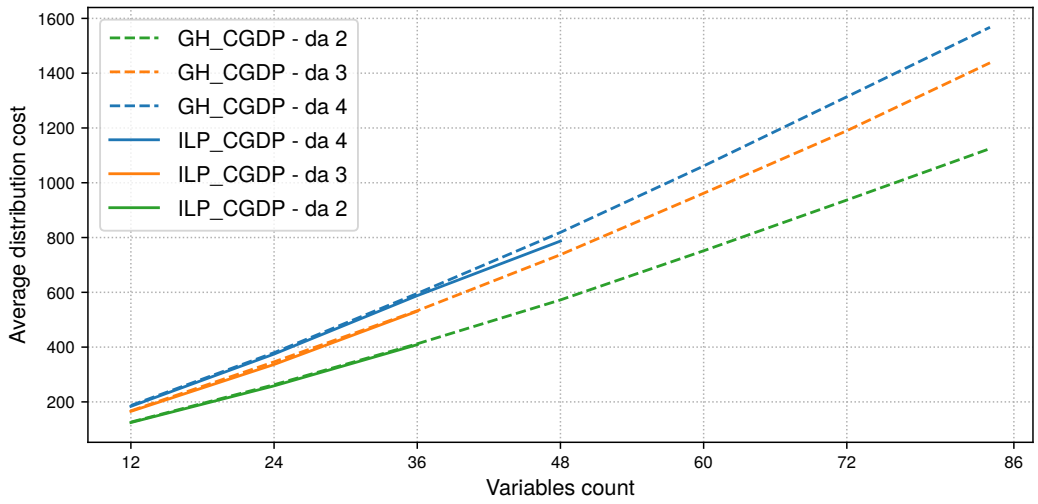


Figure 4.16 – Distribution cost for random graph coloring problems with optimal and heuristic methods, when using a constraint graph representation

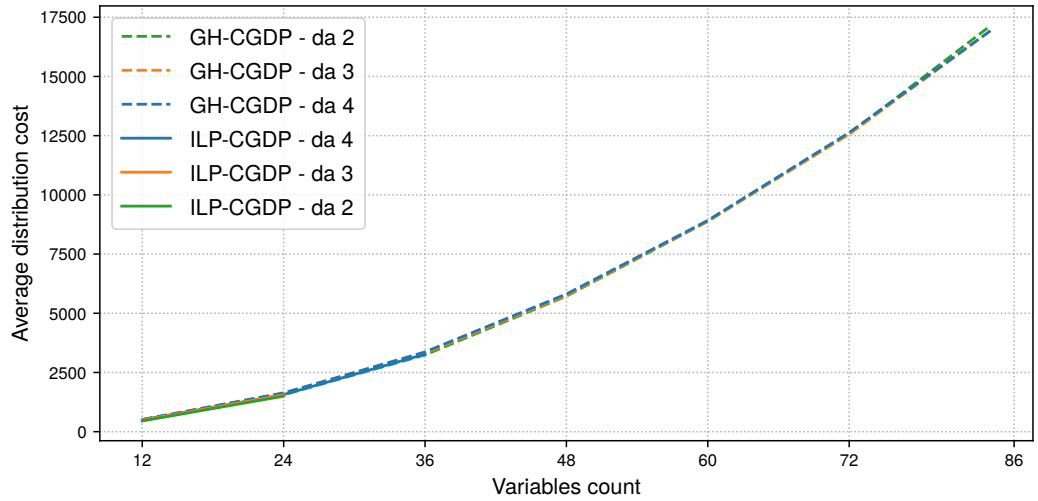


Figure 4.17 – Distribution cost for random graph coloring problems with optimal and heuristic methods, when using a factor graph representation

Figures 4.18 and 4.19 show the same metrics on scale free graph coloring instances, limited to 36 variables and with a constraint graph representation. We can see that both distribution methods behave exactly as they did on random graph coloring problems: the **ILP** based optimal distribution method can only be used on small instances, but the heuristic approach is very fast and can handle large problems, while still producing near-optimal distributions. Results are also similar when using a factor graph representation, which is why we do not include extra figures for it.

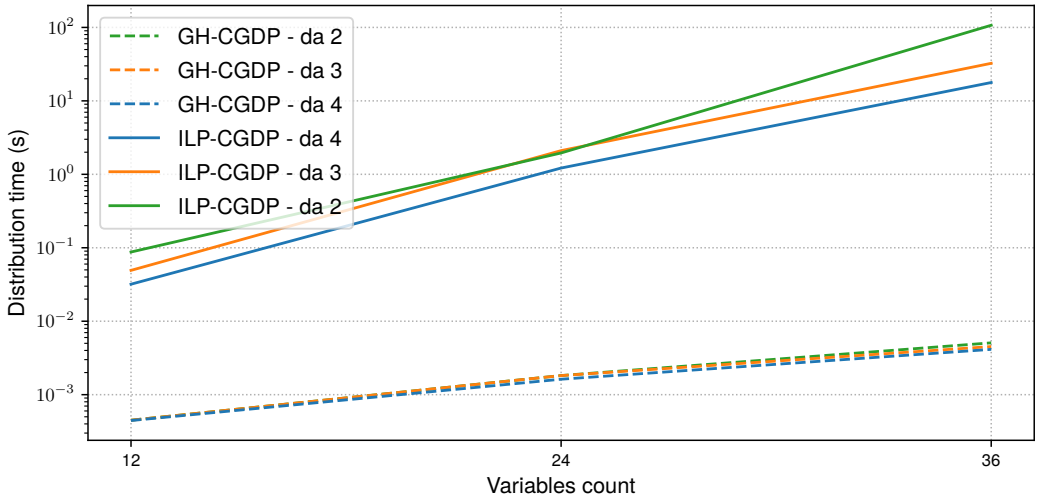


Figure 4.18 – Distribution time for scale free graph coloring problems with optimal and heuristic methods, when using a constraint graph representation

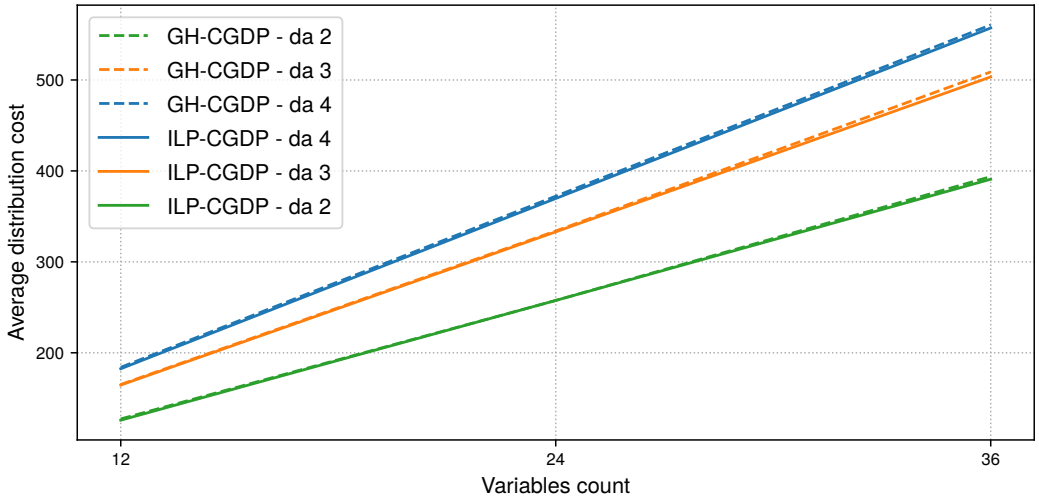


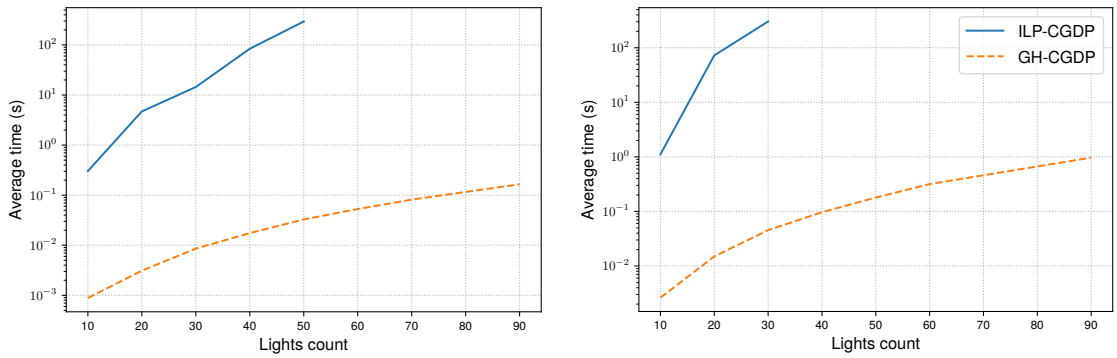
Figure 4.19 – Distribution costs for scale free graph coloring problems with optimal and heuristic methods, using a constraint graph representation

4.6.3 Evaluating Generalized Distribution on SECP

In this second experiment, we use our optimal and heuristic distribution methods on **DCOP**s representing **SECP** instances. We generate instances with increasing numbers of lights, physical models and rules, using the same protocol than described in Section 3.3.2: 20 instances are generated for each problem size and distributed using **GH-CGDP** and **ILP-CGDP** (with a 30 seconds time limit as previously), when using a constraint graph representation and a factor graph representation.

Figure 4.20 shows the time required to compute the distribution when using constraint graph (4.20a) and factor graph (4.20b) representations of the **DCOP**. We only plot optimal distribution times for problem sizes for which *all instances* could be distributed. As for standard benchmark, distribution

is harder to compute when using a computation graph based on a factor graph representation and the optimal distribution is restricted to relatively small instances while the heuristic distribution can be used on very large instances: it can easily deal with **SECP** with 90 lights and could distribute much larger instances.



(a) Constraint graph representation

(b) Factor graph representation

Figure 4.20 – Time for distributing **SECP** instances with optimal and heuristic methods

Figure 4.21 and 4.22 depict the distribution costs for our **SECP** instances when using the heuristic and optimal methods, respectively with a constraint graph and factor graph representation. As the distribution cost is made of communication and hosting costs (see Definition 15), we also plot the communication and hosting components. Notice that we include in this figure the optimal cost of distribution for all problems size for which at least some instances could be distributed. As previously, we can see that **GH-CGDP** produces very good quality distributions while requiring several orders of magnitude less time.

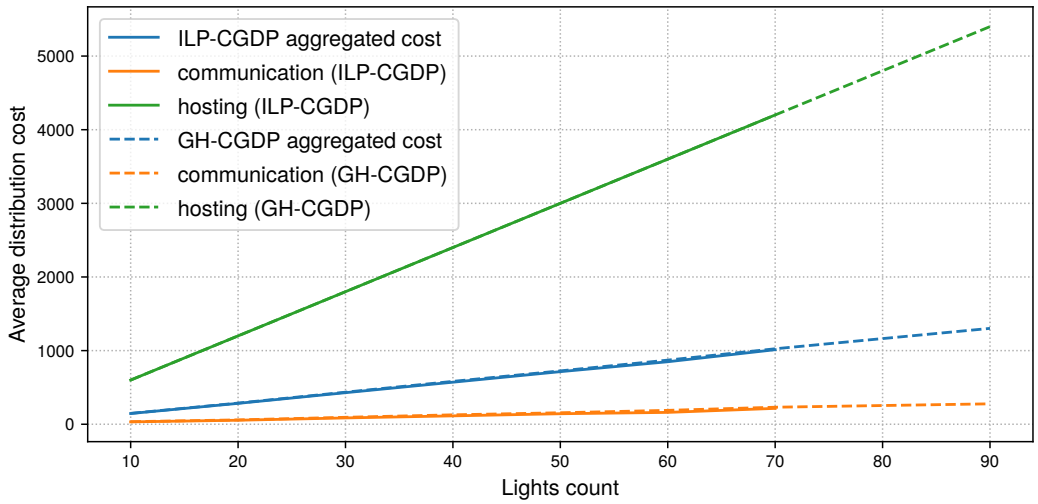


Figure 4.21 – Cost of distributions for **SECP** instances with a constraint graph representation

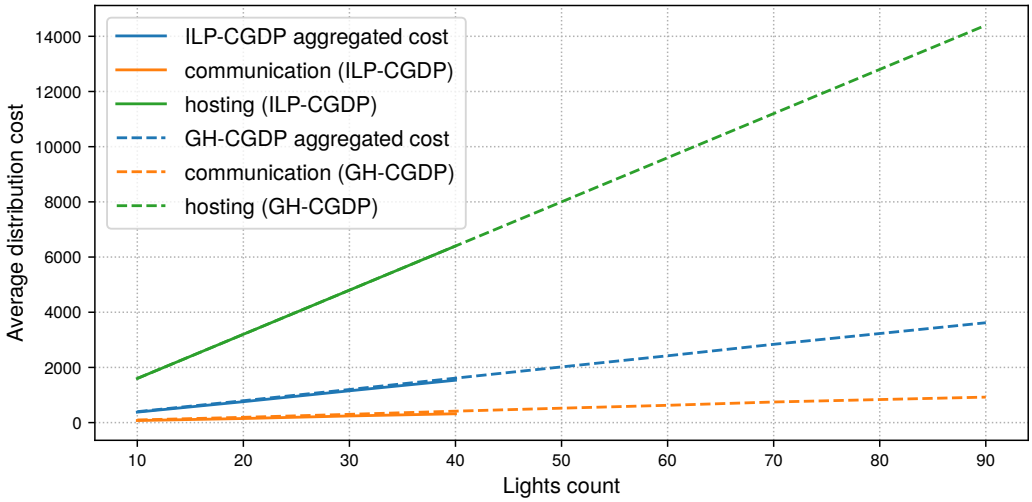


Figure 4.22 – Cost of distributions for **SECP** instances with a factor graph representation

4.7 Summary

In this chapter, we argued that distribution is a major concern when applying **DCOP** approaches on real-world problems, especially when dealing with **AmI** and **IoT** systems.

Then, we proposed several techniques for computing such distributions. Our first three techniques, **GH-SECP-CGDP** (Definition 7), **GH-SECP-CGDP** (Definition 8) and **ILP-SECP-CGDP** (Definition 10) are designed specifically for the **SECP** model introduced in Section 3.1 and are differentiated according to the graph model used by the different **DCOP** algorithms.

We also introduced the idea of a *Computation Graph* to model the more generic case of arbitrary graph models and to account for the various types of **DCOP** algorithms. Based on this model, be proposed two methods, **ILP-CGDP** (Definition 15) and **GH-CGDP** (Definition 16) for the distribution of these computations over a set of agents, focusing on characteristics of **IoT** systems: communication and devices capacities.

Based on our experimental evaluations of these distribution methods, both on **SECP** instances and classical benchmarks, we conclude that computing an optimal distribution is unrealistic in all but the smallest problems. However, the definition of an optimal definition is a very useful tool, as it gives us a metric to evaluate the quality of suboptimal distributions. Indeed, the heuristic distribution methods we devised produce very good quality results while requiring several orders of magnitude less time to compute. These methods are thus realistic even on large systems.

5 Resilient Decision-Making in Dynamic Environments

In the previous chapters, we designed the **SECP**, a model for autonomous coordination among devices in a **AmI** environment, and used **DCOP**s to implement it on real settings. Then, we discussed the need of distributing the computations required for these **DCOP**s on the devices, which we consider as agents. However, up to now our **SECP** have been defined as a static problem, which does not change during runtime. Yet, we identified in Section 4.4.3 that there are cases where the problem might change and we would need to recompute an updated distribution. In this chapter, we take things further and study the dynamic aspect of the **SECP** and devise techniques to ensure that our system adapts to these changes.

5.1 Decisions in a Dynamic Environment

The **SECP** has not been defined as a dynamic problem. However, when deploying such distributed systems in open environments like the one envisioned by **AmI** and **IoT**, it is difficult to consider that the problem, including the devices that runs it, will never change during its active lifetime. We will now see the types of changes that may happen at the **SECP** level and how they translate at the **DCOP** level.

5.1.1 SECP is a Dynamic Problem

The **SECP** is designed to model an **AmI** system where devices cooperate autonomously, without any centralized decision point, to reach user-defined objectives in a home environment. In Section 3.1.2 we defined the main components of this model: *Devices* (sensors and actuators), *Scenes* (a.k.a. *Rules*) and *Physical Models* for the environment states. Let's review for each of these elements the potential changes that could happen during the lifetime of the system.

Devices. As a home is a living environment, the set of devices available within it is also not fixed and may change relatively frequently. The most obvious reason is that users buy new devices and expect them to work seamlessly with the existing system and its configuration. Additionally, devices fail and might be replaced by new devices with different, although generally similar,

characteristics: for example when replacing a broken light bulb, a user will generally buy a new one from a more recent generation, with a better energy efficiency and thus a different e_i energy-cost function. Finally, some devices are mobile by design: for instance, a smartphone is carried by the user, meaning that it will be available in the system when the user is at home and absent otherwise.

Scenes. Another obvious reason for change in a **SECP** are the scenes, whose rules define the user objectives. During the nominal use of the system, users will generally add, remove or modify rules to better fit the system to their preferences and adapt them to potentially changing life routines. For example, An user who is coming home everyday at 5PM will have some rules that depend on that time and that will need modifications in the case this schedule is altered.

Physical Models. Changes on physical models, which represent the relation between actuators and the environment, might be less obvious but they are inevitable, nonetheless. First, when a device is added or removed, one or several physical models may need to be modified to take into account their effects. Additionally, we stated in Section 3.1.4 that online machine learning approaches could be used to improve the definition of these models over time, meaning that the definition of the model's function φ_j might evolve even without any change in the set of devices in its scope. Notice that some changes of these models can be avoided by integrating sensors variable in their scope. For example, the light level in a room clearly depends on the time of day, but we can avoid continuously changing the model by defining a *clock* sensor that represents time and defining a model function that depends on this sensor value.

Environment. Although we do not model it explicitly in **SECP**, all the aforementioned elements are connected to an environment that may also change over time. We assume here that all changes in the environment that are significant for our problem would be captured through sensor devices, which the physical models rules depends on: time, temperature, luminosity, etc. Of course, some changes may not be visible for our sensors. For example, a wall might be removed or a new window created, but for such major modifications, we argue that the system would need to be reconfigured or re-installed manually anyway.

5.1.2 Impacts of SECP dynamics at the DCOP level

As we have seen, all elements of the **SECP** might undergo changes during the nominal execution of the system. We want these changes to be taken into account automatically, without any user intervention, by the system, which should keep working normally and work toward the possibly changing user goals. We will now review how these changes at the **SECP** level translate on the **DCOP** that is used to implement it.

Devices. When a device is added or removed from a **SECP**, so is the corresponding actuator (or sensor) variable x_i and energy cost function e_i in the **DCOP**. Additionally, the agent embodied in that device is also added or removed.

Scenes. When a rule is added or removed, a corresponding constraint, represented by a utility function r_k, is added or removed as well. When a rule definition is modified, the corresponding constraint (utility function) in the **DCOP** is also modified. This modification may only involve the definition of the function but it can also include modification of its scope.

Physical Models. A physical model can undergo several kinds of changes, which translate into a change in the definition of the corresponding constraint φ_j, potentially including its scope. Additionally, when a physical model is added or removed, so are the corresponding constraint and variable used to represent that model.

5.1.3 Dyn-DCOP, a Framework for Handling Dynamics in DCOPs

One extension of the **DCOP** framework, namely **Dyn-DCOP** (see Section 2.3.2.3), deals with problems whose definition changes during execution, as is it the case in the **SECP** model. Building on **Dynamic Constraint Reasoning (DCP)** [115], a **Dyn-DCOP** is generally represented as a sequence $\{P_1, P_2, \dots P_n\}$, where each P_i is a static **DCOP** resulting from some changes in the definition of the previous one P_{i-1}.

5.1.3.1 Handling Changes in a Dyn-DCOP

The objective, when working on a **Dyn-DCOP**, is thus not to produce a single assignment that optimizes the constraints of the problem, but to maintain an up-to-date valid assignment and to update it to adapt to the changing definition of the problem. Of course, after some change, the current assignment may become invalid or incur unacceptable costs, and the system must react as quickly as possible to restore the quality of the solution. It is a continuous process, with no defined end, which runs as long as the system is used.

Two major approaches can be found in the literature to handle **Dyn-DCOP**s:

Reactive Approach. The classical approach, described as *reactive* [69, 106, 107, 152], is directly based on the model of a sequence of static **DCOP**s, where future **DCOP**s are entirely unknown. Each static **DCOP** is simply solved sequentially; any time the problem changes, the new **DCOP** is solved and the previous solution replaced. The advantage of this approach is that it can theoretically be used with any **DCOP** algorithm. One drawback of this approach is that this is only applicable if the rate of change is slow enough, compared to the time required to solve one of the **DCOP**s in the sequence, to terminate solving the problem before a new change occurs. Otherwise, the system would keep solving outdated problems and might even never produce a solution, as it restarts solving a new problem even though no solution as yet been found for the previous one. To avoid this issue, researchers have proposed algorithms that reuse information from previous **DCOP** to speed-up the search of the current one. A variant of this approach, also *reactive*, is to use **DCOP** algorithms that can dynamically adapt to the changes and keep working on the updated problem without restarting from scratch. Some works also consider the costs of switching from one solution to another and take this cost into account when selecting a solution for the next **DCOP** in the sequence. The target here is *solution stability*, which considers the change of solution in these dynamic systems and tries to minimize its effects.

Proactive Approach. Another approach, called *proactive* [50, 51, 88], is to consider that future **DCOP**s in the sequence are known in advance or that future potential changes may be at least partially anticipated. In that case, the objective is to look for solutions that are robust to these changes, that is to say solutions that require little or no changes despite the modification of the problem. Unfortunately, current solution methods for this model are either offline or too expensive to be used with anything but a very limited number of agents.

5.1.3.2 Modeling Changes in a Dyn-DCOP

In a **Dyn-DCOP**, changes can happen on any element of the tuple that defines a simple **DCOP**: $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$. These changes can thus affect the variables (addition or removal), domains (adding or removing values), constraints (addition, removal and change in the scope or definition) and agents (addition or removal). One approach to model the changes between the **DCOP**s P_i and P_{i+1} is to consider that all changes are modeled trough addition and removal of constraints [143]: domains can be seen unary constraints on variables, one can consider that a variable is automatically removed (respectively added) from the problem when it does not appears in any constraint's scope (respectively appears in a new scope) and any constraint's modification can be represented by a simultaneous removal of the old constraint and addition of the modified one.

By modelling all changes as constraint additions and removals this approach is extremely generic and abstract, which is often a good property, but may also abstract away too many domain-level information in our case. Furthermore, although generally used for **Dyn-DCOP**, this model does generally not account for the changes affecting agents. This can be explained by the assumptions (especially the agents-variable bijection, see Section 2.3.2.2) commonly used in **DCOP** research. However, as we stated in Section 4.1.1, these assumptions do not hold for our use-cases and, as a consequence, we must consider changes affecting agents. Finally, as we stress the importance of distribution when solving real-life problems with **DCOP**s, we also need to take into account the effect of these changes on the distribution.

For these reasons, in the following sections we elect to classify the changes on the **DCOP** into two categories:

- modifications at the *computation level*, that have not effect on the distribution of the computations over the agents,
- changes at the *infrastructure level* that require to recompute the distribution.

Notice that one given modification on an problem can be classified in the first or second category depending on the **DCOP** algorithm used to solve the problem. For example when using constraint graph-based algorithms, where computations represents variables, adding or removing rules in a **SECP** do not modify the set of computations and consequently does impact their distribution. On the other hand, if a factor graph-based algorithm is used, adding a rule would require adding a computation, which must be distributed on an agent.

5.2 Handling Dynamics at the Computation Level

In this section, we consider how we can deal with changes that do not impact the distribution of the computation graph. Modifications that impact the distribution of computations are studied in Section 5.3.

Several modifications of a **SECP** fall into this category and can be dealt with without revising the distribution:

- The removal of a device, as long as the computations hosted on the corresponding agent are removed at the same time. This would be often the case when removing a simple device such as a light bulb, that only host the variable representing this device.
- Changes that affects scenes and physical models without modifying their scope. This kind of change can be the result of a continuous machine learning mechanism that adapts the physical model function based on the value of the sensors.
- Adding or removing a rule, when using a **DCOP** algorithm that only defines computations for variable. When using a factor graph-based algorithm, the situation is different as this kind of change requires adding or removing computations, which must be distributed on an agent.

More generally, when looking at generic problems modeled as a **DCOP**, the following modifications do not require to revise the distribution:

- adding and agent,
- removing an agent, when all computations hosted on it are removed at the same time,
- modifying the domain of a variable or the scope of a constraint,
- adding a constraint, when using an algorithm and a graph model that does not define computations for constraints (e.g. algorithms based on a constraint graph like DSA),
- removing a constraint or a variable.

Notice that after some of these changes, it could very well be *beneficial*, depending of the use-case, to revise the distribution. It could for example allow exploiting the extra capacity offered by a new agent or improving communication costs and/or distribution costs by moving computations from one agent to another. It is, however, not *mandatory* as the previous distribution is still valid, although potentially not optimal.

5.2.1 Using the Dyn-DCOP Reactive Approach for SECP

As we only consider here modifications that do not impact the distribution, we can leverage the existing works on **Dyn-DCOP**.

In our case, we argue that we cannot predict future changes in the system and we consider that we do not really need the robustness of the solution that the self-stabilizing approach is aiming for. This characteristic is interesting when switching from one solution to another one induces a large cost to the system, as it can be the case for vehicle routing or meeting scheduling. In these cases, the solution stability is indeed primordial, and it is of paramount importance to take into

account the cost of transitioning to a new solution versus the benefit provided by this new solution. Therefore, it is potentially more interesting to trade some optimality on the solution for a smaller amount of adaptation when changes occur.

However, in an **AmI**, changing the solution is generally costless and we want our system to adapt to the environment as it is right now, and do not mind some changes. Of course, this holds as long as these changes are subtle and not too frequent –nobody wants to see the light bulbs flickering continuously! But as the changes in our environment are either relatively modest (e.g. the modification of the physical model for a room), and requires small adaptations, or abrupt and in this case require fast and major changes (e.g. when a device fails and we want to get back to the user's target state as fast as possible), we argue that reactive **DCOP** approaches are better suited to our use case.

5.2.2 Selecting Suitable Dyn-DCOP Algorithms for SECP

With such a reactive approach, there are still several considerations to take into account when selecting the right algorithm to use to solve the consecutive **DCOP**s in the **Dyn-DCOP**.

In an ideal setting, we could theoretically use any **DCOP** algorithm. However, as with **AmI** and **SECP** we consider real-world settings, we must ensure that the characteristics of this environment matches the assumptions that need to be satisfied for the algorithm to function properly.

- Many algorithms require perfect message delivery, meaning that messages must never be lost, which is not possible to guarantee in our target environments, with low-quality communication network that are generally used in **AmI** and **IoT** settings. For example, this is the case for synchronous algorithms, like **DPOP** or **DSA**, where loosing messages would freeze all nodes waiting indefinitely to move to the next synchronous round. Some other algorithms tolerate messages loss. **A-MaxSum** and **Anytime DPOP (AnyPop)** for example, do not require wait for all their neighbors' messages and keep producing approximate solution.
- Some algorithms assume that the changes from one **DCOP** to the next one in the sequence are known to all agents, which is also generally not realistic in real world settings.
- Additionally, the agents in **AmI** are generally embodied in constrained devices and cannot run algorithms that require expensive computations (memory and CPU wise), therefore we must target lightweight algorithms.
- Finally, we can afford approximate solutions as, when deciding the light level to apply to a light bulb, a user would generally not notice the difference between two solutions that are relatively close.

We now consider some algorithms developed by the **DCOP** community to address **Dyn-DCOP**s:

Self-stabilizing DPOP (S-DPOP) is a self-stabilizing [33] synchronous inference-based algorithm [107] specifically designed for **Dyn-DCOP**. **S-DPOP** is a **DPOP** variant where the three phases (`DFS`, `UTIL` and `VALUE`) are implemented using self-stabilizing protocols.

Reviewed Super-stabilizing DPOP (RS-DPOP) is an improved version [106] of **S-DPOP**, designed for optimal solution stability, which takes into account commitment deadlines during

the `UTIL` and `VALUE` phases. This allows **RS-DPOP** to work on systems that are not fully synchronized as it only requires agents to have synchronized clocks.

AnyPop is a variant [105] of **Approximative DPOP (A-DPOP)**, which is itself a variant of **DPOP**. In order to avoid large messages **A-DPOP** drops dimensions in **DPOP**'s join/project operations, resulting in a known approximation ratio. **AnyPop** builds upon this approach and allows agents to select a value without waiting messages from its neighbors, if the error bound is low enough. This provides **AnyPop** with some built-in fault tolerance; if messages are lost, the system keeps running and there is a graceful degradation of solution's quality.

MaxSum is an inference-based algorithm [36] based on belief propagation (see Section 2.4.2.3 for a full description), operating on factor graphs by performing a marginalization process of the cost functions, and optimizing the costs for each given variable. **MaxSum** can be implemented both as a synchronous and as an asynchronous algorithm, in which case we term it **A-MaxSum**. Although not specifically designed for **Dyn-DCOP**, the authors highlight that it can handle them well and produce continuously updated results, especially when implemented asynchronously.

Asynchronous Distributed Stochastic Algorithm (A-DSA) is an asynchronous version of the local search algorithm **DSA** [158] (see Section 2.4.2.1 for a full description). While in **DSA** agents proceed in synchronized rounds, **A-DSA** [40] is asynchronous, each agent evaluating *periodically* if it could improve its partial assignment. Like **A-MaxSum**, **A-DSA** has not been designed for **Dyn-DCOP** but can still be applied in these settings thanks to its asynchronous and stateless nature (see Section 5.3.2.1).

Table 5.1 summarizes the characteristics and assumptions made by these **Dyn-DCOP** algorithms. As we can see, the constraints of our target environment severely restrict our choices. In the reminder of our study, we will concentrate on approximate lightweight algorithms like **A-MaxSum** and **A-DSA**.

Table 5.1 – Some **Dyn-DCOP** algorithms assumptions and characteristics

Algorithm	Messaging	Knowledge of changes	Computation weight	Optimality
SDPOP	Perfect	Perfect	Expensive	Optimal
RSDPOP	Imperfect	Perfect	Expensive	Optimal
AnyPOP	Imperfect	Perfect	Expensive	Optimal
A-MaxSum	Imperfect	Local	Lightweight	Approximate
A-DSA	Imperfect	Local	Lightweight	Approximate

5.3 Handling Dynamics at the Infrastructure Level

Up to now, we only considered changes of the **SECP** that modify the **DCOP** without impacting the distribution of the computations that are used to actually solve that **DCOP**. In this section, we study the modification of the infrastructure, that is to say, changes that impact the distribution. Such modifications require to fix the distribution to ensure that all needed computations still run. We will overview several solutions that can be used to achieve this goal.

5.3.1 Dynamics that Impact the Distribution of a DCOP

As we listed in Section 5.2 the modifications that the problem can undergo while keeping the same distribution, we now list modifications that impact the distribution:

1. Any modification that requires adding a new variable to the **DCOP** will also require adding a new computation, meaning the distribution must be revised to place this computation on an agent.
2. When using an algorithm that defines computations for constraints, adding a new constraint also require revising the distribution.
3. Removing an agent, when some of the computations hosted on it must be preserved, always makes the distribution invalid. This is generally the case when hosting some *shared* computations (see Section 4.1.3). These computations represent a part of the definition of our problem and thus must be moved to another agent.
4. When adding an agent, revising the distribution is generally not mandatory: we can simply ignore it and the system will keep running with the previous set of agents. It can however be beneficial to review the distribution to benefit from the extra capacities and potentially reduce communication and hosting costs.

In the case of **SECP**, modifications (1) and (2), namely the introduction of a new constraint or of a new variable without a corresponding device (i.e. not an actuator variable), can only be the consequences of the addition of new models and/or rules in the system. Although these modifications impact the distribution, they do not really qualify as *infrastructure changes* and can be dealt relatively easily, therefore we do not elaborate on these cases and will concentrate on changes at the infrastructure level.

- Adding a rule requires an user interaction, which is typically done through a generally powerful dedicated device, for instance a home computer or a tablet. In that case, we can rely on this device to compute a revised distribution using any of the optimal or heuristic centralized methods discussed in Section 4.
- Adding a new physical model only makes sense when the user specifies a new rule (otherwise the model is useless) with a new physical model (related to a new sensor) he has obtained. For instance, a user installs a sound level sensor and adds a new rule which exploits the sound level somehow. Such a situation, once again, only occurs when the user interacts through his dedicated device with the system. Thus, distribution can be done in a centralized way, as in Section 4.

Although these modifications impact the distribution, they do not really qualify as *infrastructure changes* and can be dealt relatively easily, therefore we do not elaborate on these cases and will concentrate on changes at the infrastructure level.

On the other hand, modifications (3) and (4) (i.e. arrival and departure of agents) modify the set of available agents and therefore really impact the infrastructure used to run the computations that solve the **DCOP**. As these modifications may happen at any time, and generally not during user interaction, we cannot rely on a powerful device to run our centralized distribution solution

methods. When such appearance and disappearance occur, the devices have to self-adapt without help of a central computer. As a consequence, we will concentrate on these cases.

5.3.1.1 Handling Agent Departure

As discussed previously, in an open system agents may leave at any time. Such agent may host shared computations that must be preserved and moved to another agent for the system to function properly. We term *orphaned computations*, the computations that were hosted on a departed agent and must be migrated to a remaining agent. When considering agent departure, we can identify two cases:

- *Safe removal*, which happens when the device leaves the system voluntarily. On such event, the agent can actively migrate computations before leaving and we can assume that the definition and run-time state of these computation will not be lost.
- *Unsafe (or unexpected) removal*, which happens when an device fails (or is simply disconnected unexpectedly). In that case, the device obviously cannot migrate its shared computation, resulting in orphaned computation.

Unsafe removal is more critical, we propose two methods for dealing with such change. In Section 5.4 we present a technique that build upon **ILP-CGDP**, the centralized mechanism for optimal distribution introduced in Section 4.5, and adapt it for local repair. In Section 5.5 we study how we can make the system resilient to simultaneous disappearance of several agents.

5.3.1.2 Handling Agent Arrival

When an agent enters the system, there are two situations to consider:

(a) *The Participating Agent*, when the agent already hosts some computation(s) linked to other computation(s) in the system. In the case of **SECP** this can for instance be a new light bulb, which hosts the variable-computation representing its emitted light level. Such computation will eventually be connected to the physical model(s) representing the area where the new device is located, and to a rule, if the user set some specific target for this light bulb.

(b) *The Blank Agent*, when the agent does not host any computation and may only be used to offload existing computations already hosted on other agents.

In Sections 5.6.1, 5.6.2 and 5.6.3 we will present several techniques to deal with these two cases.

5.3.2 Prerequisites for Handling Infrastructure Changes

Before diving into solution methods, we first need to expound the prerequisites that must be satisfied to be able to deal with infrastructure changes.

5.3.2.1 Discovery

Revising a distribution implies moving computations from one agent to another. As computations on agents communicate with one another, agents have to know to which other agent they send messages. Such requirement depends upon a discovery mechanism which ensures that an agent

can publish what computation it hosts and that one can discover which other agent is hosting a computation it must communicate with.

This discovery mechanism must also provide information on new device arrival and devices departure or failure. As departure detection is commonly implemented using *keepalive* messages, one can generally not assume that each agent will be informed of any device failure in the system; as the system might be quite large, only neighbor devices are commonly monitored.

Distributed discovery protocols providing these features already exist. For example, mDNS [58] and DNS-SD [57] are generally used in current IP-based **SHE** systems while **Constrained Application Protocol (CoAP)** Discovery [55] and **Constrained RESTful Environments (CoRE)** Resource Directory [56] have been designed for constrained nodes and networks like for instance **6LowPan (IPv6 over Low-Power Wireless Personal Area Networks)**.

In the remainder of this document we will not elaborate further on this topic and assume that such discovery mechanism is available in the system. We also assume that this mechanism allows an agent to monitor the presence of agents in its neighborhood and be informed of their departure. However, agents are not aware of the disappearance of agents that they don't share any link with.

5.3.2.2 Preserving the Problem Definition

When representing a problem as a **DCOP**, which is implemented using a computation graph, the computations collectively encode the definition of the problem itself. As we are aiming for fully distributed and decentralized systems, where no central authority is available to restore these definitions after a change in the system or the failure of a device, we must ensure that these definitions are not lost in such events.

Additionally, when a computation is moved from one agent to another, the target agent needs to know the definition of that computation in order to be able to instantiate it. This is particularly important when handling agent failure; by definition these are not planned and the failed agent is not available any more to communicate the definition to the new host.

These definitions can be given during the initial deployment and revision phases. We will present several approaches in the next sections.

5.3.2.3 Maintaining the Solving Process State

In addition to the problem definition, another type of information might be lost when agents leave the system: the state of the solving process. As a matter of fact, when solving a **DCOP**, computations exchange messages but also generally keep an internal state that represents their current knowledge and is built from the messages they received from their neighbors. When an agent fails, such state is lost, which hinder the solving process. For instance an algorithm that relies on information acquired and stored by agents (like *nogoods* in **Asynchronous Backtracking (ABT)** or *costs* in **ADOPT**) might even not be able to restart after a reparation.

We can identify two approaches to address this issue:

(a) keep one or several copies of that state, in order to be able to restore it when restarting the computation on another agent;

(b) use stateless or *almost-stateless* algorithms, where this state can easily be discarded.

Option (a) is similar to what is required for computation definitions. However, these internal states may be relatively large. In **DPOP** for example, the internal state of a computation is made of a n-ary relation whose size depends on the width of the sub-tree rooted at that point, and the size of the domain of the variables, which might lead to relatively big n-dimensional hypercubes. Another issue is that the copies of these states must be updated regularly. Indeed, every time a computation receives a message, it might update its internal state and when restarting a computation after migration, we want it to be restored with a state that matches the one expected by its neighbors. This update process can be complex and requires a lot of communication, which is not suitable for the constrained networks available in our target environments.

Option (b) is thus much better suited in our case, but severely restricts the **DCOP** algorithms we might use in our system. Very few **DCOP** algorithms are stateless; **DSA** and **MGM** can be considered as stateless as they receive at each cycle the current value (respectively cost) from their neighbors. A computation does not even need to store this information, as it only takes its decision based on the messages received in the current cycle. However, this only stands as long as these algorithms are implemented synchronously. On the other hand, we consider **A-DSA** to be *almost-stateless*. As there is no cycle in an asynchronous algorithm, the computation cannot assume it will receive updated value messages from all its neighbors and most store the value received in the last messages. However, in case that information is lost it will be reconstituted as soon as a message has been received from each of its neighbors. At that time, the computation can restore its nominal behavior and is not affected by the information loss anymore. This is also the case for **A-MaxSum**, where the accumulated costs received from neighbors can be rebuild.

5.4 Migrating Computations in the Neighborhood

Now that we have seen the two different types of dynamics and presented the prerequisites for handling infrastructure change, we introduce in this section a solution method to deal with agent's disappearance by considering it as a local problem.

This approach re-uses **ILP-CGDP**, the linear program for computation graph distribution, presented in Section 4.5. However **ILP-CGDP** is a centralized approach, and when repairing no single agent is identified to solve the problem. Moreover, even if we somehow selected one agent to be responsible for solving this **ILP**, that task would be too computationally intensive to be solved on our constrained devices. In order to make the problem easier, we elect to restrict it to the neighborhood of the departed agent.

Practically, we consider adapting the deployment of the computation graph locally, by only considering a reduced set of agents (termed neighborhood) and a portion of the computation graph (set of computations hosted by the neighbors). It is a local and heuristic approach: the resulting distribution might not be optimal, but potentially requires far less computation and still allow to repair the system and avoid loosing any shared computation.

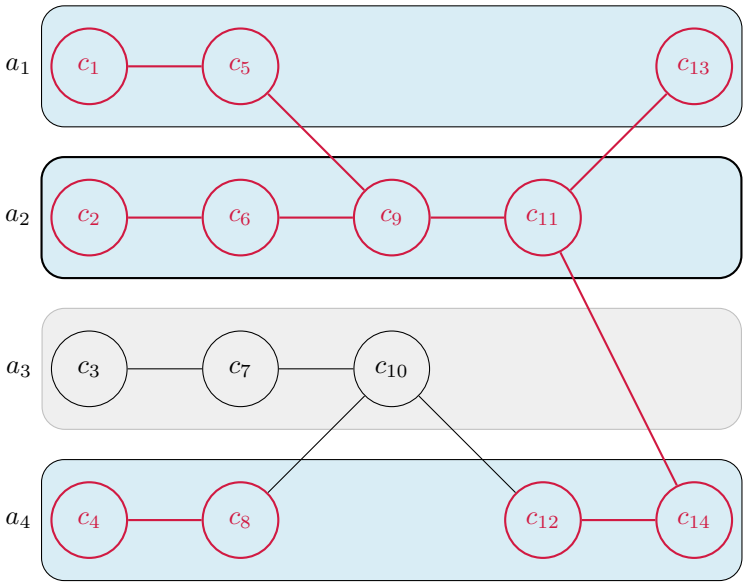


Figure 5.1 – Representation of the neighborhood of agent a_2 in a computation graph

5.4.1 Definition of Neighborhood

Let's define formally the notion of neighborhood as follows:

Definition 17 (Neighborhood)**.** *Given the current distribution ν, the* neighborhood $\mathcal{A}[a_k]$ *of an agent a_k is defined as follows:*

$$\mathcal{A}[a_k] = \{a_\ell \mid \exists (c_i, c_j) \in E, \nu(c_i) = a_k,\ \nu(c_j) = a_\ell\} \cup \{a_k\}$$

if the agent a_k hosts at least one computation, and $\mathcal{A}[a_k] = \mathcal{A}$ otherwise.

Similarly we define $E[a_k]$, the set of edges connected to the neighborhood:

$$E[a_k] = \{(c_i, c_j) \mid \nu(c_i) \in \mathcal{A}[a_k],\ \nu(c_j) \in \mathcal{A}[a_k]\}$$

And $\mathcal{C}[a_k]$, the set of neighborhood computations:

$$\mathcal{C}[a_k] = \{c_i \mid (c_i, c_j) \in E[a_k]\}$$

Example 13. *Figure 5.1 pictures this notion of neighborhood for agent a_2 in a computation graph representing a **SECP** distributed on 4 agents, with the three sets $\mathcal{A}[a_2]$, $E[a_2]$ and $\mathcal{C}[a_2]$.*

The sets of neighborhood computations and edges in the neighborhood, respectively $\mathcal{C}[a_2]$ and $E[a_2]$, are depicted in red. The set of neighbor agents of a_2, depicted in blue, is composed of agents $\{a_1, a_2, a_4\}$.

5.4.2 Restricting the ILP-based Distribution

Based on these definitions, we can define a cut version of **ILP-CGDP** (Definition 15), restricted to the sub-graph of the computation graph defined by the neighborhood of the departed agent. When an agent a_k fails, the distribution problem is then to decide, for each orphaned computation, which agent in $\mathcal{A}[a_k]$ should host that computation.

Thus, we rewrite a cut version of **ILP-CGDP**, restricted to the set defined by the neighborhood $\mathcal{A}[a_k]$, $\mathcal{C}[a_k]$ and $E[a_k]$ as follows:

$$\begin{aligned}
\underset{c_i^m}{\textbf{minimize}} \quad & \omega_{\mathbf{com}} \cdot \sum_{(i,j)\in E[a_k]} \sum_{(m,n)\in\mathcal{A}[a_k]^2} \mathbf{com_a}(c_i, c_j, a_m, a_n) \cdot \alpha_{ij}^{mn} \\
+ \quad & \omega_{\mathbf{c_{host}}} \cdot \sum_{(c_i,a_m)\in\mathcal{C}[a_k]\times\mathcal{A}[a_k]} c_i^m \cdot \mathbf{cost}(a_m, c_i) \qquad (5.1)
\end{aligned}$$

subject to

$$\forall a_m \in \mathcal{A}[a_k], \quad \sum_{c_i\in\mathcal{C}} \mathbf{mem}(c_i) \cdot c_i^m \leq \mathbf{w_{max}}(a_m) \qquad (5.2)$$

$$\forall c_i \in \mathcal{C}[a_k], \quad \sum_{a_m\in\mathcal{A}[a_k]} c_i^m = 1 \qquad (5.3)$$

$$\forall c_i \in \mathcal{C}[a_k], \quad \alpha_{ij}^{mn} \leq c_i^m \qquad (5.4)$$

(5.5)

$$\forall c_j \in \mathcal{C}[a_k], \quad \alpha_{ij}^{mn} \leq c_j^m \qquad (5.6)$$

(5.7)

$$\forall c_i, c_j \in \mathcal{C}[a_k], a_m \in \mathcal{A}[a_k], \quad \alpha_{ij}^{mn} \geq c_i^m + c_j^n - 1 \qquad (5.8)$$

(5.9)

Definition 18 (ILP-CGDP$[a_k]^-$)**.** ***Integer Linear program for CGDP restricted to the neighborhood (ILP-CGDP**$[a_k]^-$**) consists in ILP-CGDP (Definition 15) restricted to the set of agents** $\mathcal{A}[a_k]\backslash a_k$ *and to the computation graph* $\langle \mathcal{C}[a_k], E[a_k]\rangle$.

Note that **ILP-CGDP** optimizes the distribution for hosting and communication costs. When applying it to a **SECP**, we force actuators' variable and cost to be hosted on their device by assigning an infinite cost for all other agents.

Example 14. *On the SECP depicted on Figure 5.1, when agent* a_2 *fails, the shared orphaned computations* c_9 *and* c_{11} *must be migrated to one of the agents in the neighborhood, namely* a_1 *or* a_4*. Thus, in this case, ILP-CGDP*$[a_k]^-$ *only involves two agents and communication costs with the three computations that communicates with the two orphaned computation, namely* c_5*,* c_{13} *and* c_{14}.

5.4.3 Solving ILP-CGDP$[a_k]^-$

This problem can be solved either by one agent (if the size of the problem is not too large) or by the agents composing the neighborhood. In both cases it only requires local and limited knowledge on

the global **DCOP**, which makes it ideal for large and complex systems.

In the distributed solving case, several distributed optimization techniques could meet the requirements like the distributed simplex method designed for multi-agent assignments [19], keeping exactly the same encoding as **ILP-CGDP**, or dual decomposition methods like the efficient AD^3 method [81], that requires **ILP-CGDP** to be encoded using tractable high order potentials [139], which is possible for of all constraints in **ILP-CGDP**, and then implement a distributed decoding of the LP relaxation to assign integer values to decision variables. However, while providing good optimality, both distributed simplex and AD^3 may require several rounds (thus message exchanges) to reach good quality solutions. For instance, from a conjecture in [19], the average time complexity of this technique is linear in the diameter of the graph ($\mathcal{O}(diam(FG))$), with polynomial communication load. In **SECP** case, the diameter of the FG is not bounded but mainly depends on the number of rules and models, and their interdependencies. In the case of real smart home settings, models and rules will mostly influence local areas (rooms, floor, etc.) and interdependencies, thus diameters, will be limited.

5.4.4 Limitations of ILP-CGDP$[a_k]^-$-based Solution

While this approach provides a working solution to repair the system when one of the agents fails, it has some limits, which we will now discuss:

Computation Definitions. As mentioned in 5.3.2.2, in order for the selected agents to be able to instantiate and run the migrated orphaned computations, they need to known their definitions. For that purpose we assume in this approach that during the initial distribution, each agent is provided with a copy of the definition of all computations in its neighborhood. This means that each agent may need to know, and store, a large number of computation definitions and may even keep definitions for computations they cannot host anyway, because they have already reached their capacity limit.

Neighborhood. In this approach, the orphaned computations must always be migrated to an agent of the neighborhood, which could prove impossible if the neighboring agents of a disappearing one don't have enough capacity all together. In this case, the neighborhood could be extended by neighbors of neighbors until memory is sufficient, but that would require communicating the orphaned computations as well.

Complexity. While there are solutions methods, both centralized or distributed, available to solve the cut version of **ILP-CGDP**, these solutions are still relatively computationally intensive. Moreover it is not guaranteed they could always be applied on systems composed of constrained devices. Besides, the communication load, polynomial for the distributed simplex [19], would probably prove to be problematic on constrained networks.

Single Agent Repair. This approach is limited to a single agent leaving the system. If several agents fail simultaneously, we have no guarantee that the system can be restored. As a matter of fact, by only placing computation definitions on neighbor agent(s), we might very well loose all

definition for a computation if all the agents that possess the definition of a computation fail at the same time.

In the next session, we will devise a distributed repair method that address these limitations.

5.5 Surviving the Simultaneous Departure of Several Agents

As we have seen, the previous approach has some shortcomings –one of the most important is that is does not allow the system to survive the simultaneous failure of several agents, which may very well happen in real-world settings.

For these reasons, we introduce here the concept of k-resilience and design another approach to cope with these limitations.

5.5.1 k-Resilience

We define the notion of k-resilience as the capacity for a system to repair itself and operate correctly even when up to k agents disappear. This means that after a recovery period, all computations must be active on exactly one agent and communicate one with another as specified by the computation graph $G_{\mathcal{C}}$.

Definition 19. *Given a set of agents $\mathcal{A}$, a set of computations $\mathcal{C}$, and a distribution μ, the system is k-**resilient** if for any $F \subset \mathcal{A}, |F| \leq k$, a new distribution $\mu' : X \to \mathcal{A} \backslash F$ exists.*

One pre-requisite to k-resilience is to still have access to the definition of every computation after a failure. One approach is to keep k *replicas* (copies of definitions) of each active computation on different agents. Provided that the k replicas are placed on different agents, no matter the subset of up to k agents that fails there will always be at least one replica left after the failure, as classically found in distributed database systems [94]. Here, we apply these ideas except we keep replicas of computation definitions instead of data records, which implies that computations must be *stateless* or that their state must be restorable (*almost stateless*).

We note $\rho(c_i)$ the set of agents that possess a replica for computation c_i. In a k-resilient system, each computation has k replicas, which are placed on agents that do not host the active version of the computation:

$$\begin{aligned} \forall c_i \in \mathcal{C}, \quad & |\rho(c_i)| = k \\ & \nu(c_i) \notin \rho(c_i) \end{aligned}$$

Let's note that given the capacity constraints on the agents, keeping k replicas is not enough to warrant k-resilience and there might be no possible distribution. The maximum k value for which k-resilience can be achieved depends on the system and especially on agent's capacities. Additionally, after departure of some agent(s), the k-resilience characteristic of the repaired system should be restored, as long as there are enough nodes available.

5.5.2 Replication of Computation Definitions

The problem of assigning replicas to hosts could be considered as an optimization problem, close to Definition 13. Ideally, we should optimize replica placement for communication and hosting costs. This would ensure that when agents fail, replicas are available on good candidate agents. However, the search space for this optimization is prohibitively large:

- In a k-resilient system with n agents, there is $\sum_{0<i\leq k} \binom{n}{i}$ potential failure scenarios as we consider case where up to k agents out of n can fail simultaneously.
- With m computations, the number of possible *replica configurations* is $m \cdot \binom{n}{k}$, as we must select k agents to host the replicas for each of the m computations.
- Then, for each of these replica configurations, there are m^k *activation configurations*, as exactly one of the k replicas must be activated for each orphaned computation.

More practically, the problem of optimally distributing the k replicas of each computation on a given set of agents having different costs and capacities can be cast into a **Quadratic Multiple Knapsack Problem (QMKP)** (see [131]), which is NP-hard.

Assuming we could compute the cost of all these activation configurations, it would still not be obvious which replica placement would be better: one could consider the one allowing the best activation-configuration, or the one allowing, on average, good quality activation configurations or even the one giving the best activation configurations over the set of possible failure scenarios.

Obviously, defining the optimality for replica placement is very problem dependent. Thus, given that complexity, we opt for a distributed heuristic approach, described in the next section.

5.5.3 Distributed Replica Placement Method

We propose here a distributed method, namely **Distributed Replica Placement Method (DRPM)**, to determine the hosts of the k replicas of a given computation x_i. **DRPM** is a distributed version of iterative lengthening (uniform cost search based on path costs) with minimum path bookkeeping to find the k best paths. The idea is to host replicas on closest neighbors with respect to communication and hosting costs and capacity constraints, by searching in a graph induced by computations dependencies.

It outputs a distribution of k replicas (and the path costs to their hosts) with minimum costs over a set of interconnected agents. If it is impossible to place the k replicas, due to capacity constraints, **DRPM** places as much computations as possible and outputs the best resilience level it could achieve.

One hosting agent, called *initiator*, *iteratively* asks each of its lowest-cost neighbors, in increasing cost order, until all replicas are placed. Candidate hosts are considered iteratively in increasing order of cost, which is composed of both communication cost (all along the path between the original computation and its replica) and the hosting cost of the agent hosting the replica.

This approach is based on the assumption that the initial distribution, computed using one of the methods introduced in Section 4, is optimal or at least of good quality. As a matter of fact, if the initiator agents fails, its orphaned computation will necessarily be migrated to one of the agents that

possess its definition (i.e. that hosts one of its replicas). Therefore, by placing replicas on agents that have minimal communication and hosting cost compared to the initiator agent, we ensure that the computation will only be migrated to agents that favor a good quality distribution.

Let's first define the graph specifying the communication costs which will be developed during the search process:

Definition 20 (**route**–graph)**.** *Given a computation graph $\langle \mathcal{C}, E_{\mathcal{C}} \rangle$ and a set of agent $\mathcal{A}$, the* **route***–graph is the edge-weighted graph $\langle \mathcal{A}, E, w \rangle$ where*

- *$\mathcal{A}$ is the set of vertices*
- *E is the set of edges with $E = \{(a_m, a_n) | \ \exists (c_i, c_j) \in E_{\mathcal{C}},$ and $\nu(c_i) = a_m, \nu(c_j) = a_n\}$*
- *$w : E_{\mathcal{C}} \to \mathbb{R}$ is the weight function $w(a_m, a_n) = \mathbf{route}(m, n)$.*

Example 15. *Figure 5.2 depicts a* **route***–graph for a computation graph distributed over 4 agents.*

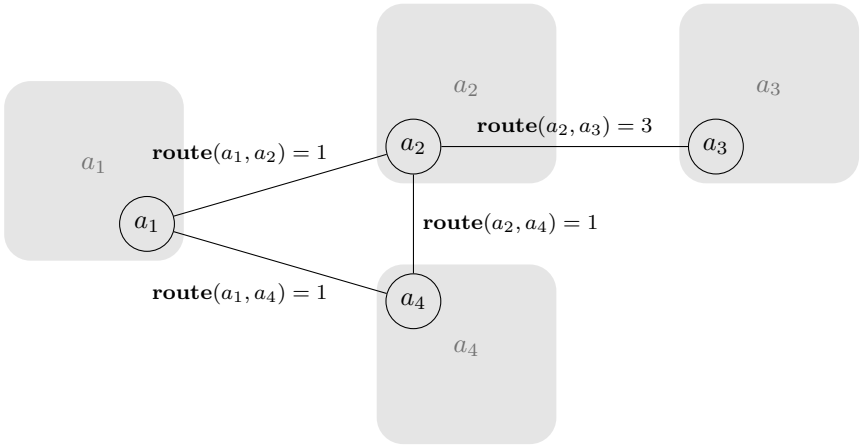


Figure 5.2 – A sample **route**–graph with 4 agents (in gray)

As to take into account both communication and hosting costs in the path costs, the **route**–graph is extended into a **route**+**host**–graph with extra leaf vertices attached to each agent in the neighboring graph, except the original host of the computation, with an edge weighted using the hosting cost of the agent, as in Figure 5.3.

Definition 21 (**route**+**host**–graph)**.** *Given a* **route***–graph $\langle \mathcal{A}, E, w \rangle$ and a computation c_i, the* **route**+**host***–graph is the edge-weighted graph $\langle \mathcal{A}', E', \mathbf{cost} \rangle$ where*

- *$\mathcal{A}' = \mathcal{A} \cup \widetilde{\mathcal{A}}$ is the set of vertices where $\widetilde{\mathcal{A}} = \{\tilde{a}_m | a_m \in \mathcal{A}, a_m \neq \nu(c_i)\}$ is a set of extra vertices (one for each element in $\mathcal{A}$ except the host of c_i),*
- *$E' = E \cup \{(a_m, \tilde{a}_m) | \tilde{a}_m \in \mathcal{A}\}$ is the set of edges*
- *$\mathbf{cost} : E' \to \mathbb{R}$ is the weight function s.t. $\forall a_m, a_n \in \mathcal{A},\ \mathbf{cost}(a_m, a_n) = w(a_m, a_n)$, $\forall \tilde{a}_m \in \widetilde{\mathcal{A}},\ \mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, c_i)$.*

Example 16. *Figure 5.3 depicts a* **route**+**host***–graph for a computation graph distributed over 4 agents.*

Notice that agent a_1 has not extra vertex $\tilde{a}_1$, representing its hosting cost, as the **route**+**host***–graph on this figure is designed to place the replicas of the computations c_i hosted on a_1 (which*

must obviously be placed on other agents).

As a matter of fact, when placing replicas, the **route+host**–*graph is specific to an initiator agent and a computation. A different* **route+host**–*graph is expanded by each agent a_k and for each of the computation c_i hosted on a_k, to place the replica for computation c_i.*

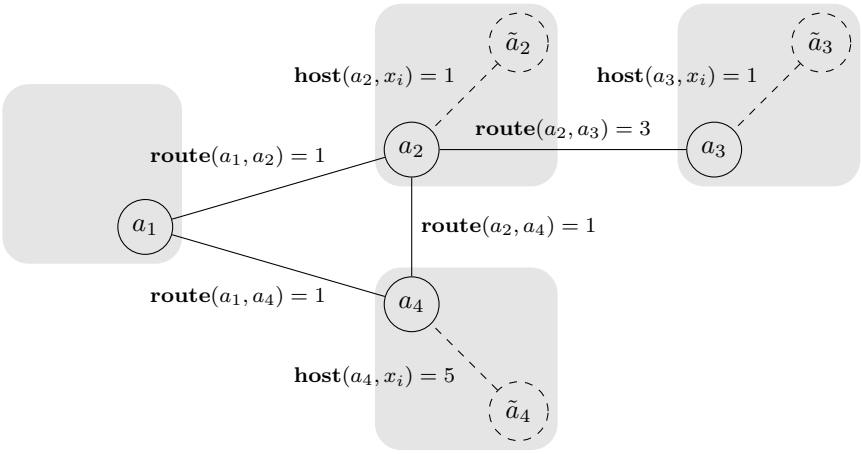


Figure 5.3 – A sample **route+host**–graph with 4 agents (in gray)

This **route+host**–graph is a search graph, expanded at runtime and explored for a particular computation c_i. Each agent operates as many instances of **DRPM** as computations to replicate over several **route+host**–graphs.

For a given **route+host**–graph , each agent may encapsulate two vertices (one in $\mathcal{A}$ and its image in $\widetilde{\mathcal{A}}$) and may receive messages concerning their two vertices, and even self-send messages.

Additionally, when assessing if an agent can host a replica for c_i, we ensure that it only accepts if it has enough capacity to activate any subset of size k of its replicas, using a predicate named `can_host?`. Of course this constraint is stronger than what might be actually needed, so, this distribution is not optimal with respect to hosting cost, since one agent might reject hosting a computation whilst it may finally have enough memory to host it. Even communication-wise, the algorithm may result on a suboptimal distribution. However, if `can_host?` is provided by an oracle or if memory is not a real constraint, and replica placement only concerns one computation, the distribution would be optimal with respect to communication and hosting costs, since our algorithm implements an iterative lengthening search [119, p.90].

DRPM makes use of two message types, `REQUEST` and `ANSWER`, with the same fields $\langle$`current`, `budget`, `spent`, `known`, `visited`, `k`, $c_i\rangle$:

1. `current`: path of the request, as a list containing all vertices messages that have been passed through from the initiator vertices to the one receiving the current message.
2. `budget`, `spent`: remaining budget for graph exploration and budget already spent on the current path,
3. `known`: map assigning cost to already discovered paths to unvisited vertices which bookkeeps the cheapest paths so far,
4. `visited`: list of already visited vertices,
5. `k`: the remaining number of replicas to host.

6. c_i: computation that must be replicated,

At the beginning, the agent requiring a computation replication initializes `known` with the paths to its direct neighbors in the $\mathbf{route}$+$\mathbf{host}$–graph and sends itself a `REQUEST` message with a budget equals to the cheapest known path. Then, agents handle messages according to Algorithms 1 and 2. The protocol ends when all possible replicas have been placed (at most k).

When receiving a `REQUEST` message (Algorithm 1), either the agent can host a replica (lines 2-10), and thus decreases the number of replicas to place, or forwards the request to other agents (lines 11-22). In the first case, if all replicas have been placed, the agent answers back to its predecessor (line 9). When looking for other agents to host replicas, if there exists a minimum cost known path starting with the currently explored path which is reachable with the current budget, the agent forwards the request to its successor in this path (with an updated cost and budget, line 16). If there is no such path, the agent fills out the map of known paths with new paths leading to its neighbors in the $\mathbf{route}$+$\mathbf{host}$–graph, when they improve the existing known paths, and sends this back to its predecessor so that it will explore new possibilities (line 22).

When receiving an `ANSWER` message (Algorithm 2), the message can either notify that all replicas have been placed (lines 1-6) or that there exists at least one replica left to place. In the former case, if the agent is the initiator, it terminates the algorithm, whilst having all the requested replicas placed (line 3), otherwise it forwards the answer back to its predecessor, until it reaches the initiator (line 6). In the later case, if the agent is the initiator it increases the budget and send a request to the closest neighbor (line 14) if any; if there is no such neighbor left, that means that there is no more path to explore and that all replicas cannot be placed, therefore the agent terminates (line 16). If the agent is not the initiator, but there exists some reachable path within current budget, it requests replication to its successor in the best known path, as when handling `REQUEST` messages (line 22). Finally, if there is no such path, it simply forwards the answer to its predecessor in the current path (line 24).

Example 17. *Figure 5.4 represents the execution of* ***DRPM*** *when agent a_1 places 2 replicas for computation c_i. The different colors in the edges depicts the path explored when increasing the budget.*

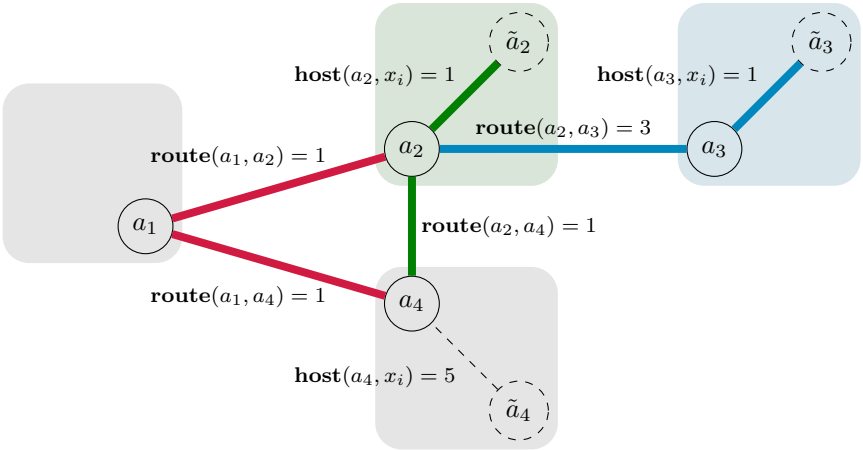


Figure 5.4 – Sample execution of **DRPM** for placing two replicas for a computation x_i

Algorithm 1: Handler for REQUEST messages

Data: current, budget, spent, known, visited, k, c_i

known $\leftarrow$ known $\setminus$ current
if me $\notin$ visited **then**
    visited $\leftarrow$ visited $\cup\{$me$\}$
    if *can_host?*$(computation_i)$ **then**
        k $\leftarrow$ k $- 1$
        add x_i to memory
        if $k = 0$ **then**
            $a_p \leftarrow$ predecessor of me in current
            send ANSWER(current, budget+**cost**(me, a_p), spent-**cost**(me, a_p), known, visited, k, c_i) to a_p
            return
$p \leftarrow \operatorname{argmin}_{e \in \{\text{paths in known starting with current}\}}$ known$[e]$
if $p \neq \emptyset$ **then**
    $a_n \leftarrow$ successor of me in p
    if **cost**(me, a_n) $\leq$ budget **then**
        current $\leftarrow$ current + a_n
        send REQUEST(current, budget-**cost**(me, a_n), spent+**cost**(me, a_n), known, visited, k, c_i) to a_n
        return
foreach $a_n \in \{a_m \mid (a_m, \text{me}) \in E', a_m \notin \text{visited}\}$ **do**
    if spent + **cost**(me, a_n) $< \min_{e \in \{\textit{paths in known leading to } a_n\}}$ known$[e]$ **then**
        known[current + a_n] $\leftarrow$ spent + **cost**(me, a_n)
$a_p \leftarrow$ predecessor of me in current
send ANSWER(current, budget+**cost**(me, a_p), spent-**cost**(me, a_p), known, visited, k, c_i) to a_p

Algorithm 2: Handler for `ANSWER` messages

Data: `current`, `budget`, `spent`, `known`, `visited`, `k`, c_i

```
if k = 0 then
    if me is root of current path then
        terminate with target number of replicas placed
    else
        a_p ← predecessor of me in current
        send ANSWER(current, budget+cost(me, a_p), spent-cost(me, a_p), known,
         visited, k, c_i) to a_p
else
    p ← argmin_{e∈{paths in known starting with current}} known[e]
    if me is root of current path then
        if p ≠ ∅ then
            budget ← budget + known[p]
            a_n ← successor of me in p
            current ← current + a_n
            send REQUEST(current, budget-cost(me, a_n), spent+cost(me, a_n),
             known, visited, k, c_i) to a_n
        else
            terminate with fewer replicas than requested
    else
        if p ≠ ∅ then
            a_n ← successor of me in p
            if cost(me, a_n) ≤ budget then
                current ← current + a_n
                send REQUEST(current, budget-cost(me, a_n), spent+cost(me, a_n),
                 known, visited, k, c_i) to a_n
        a_p ← predecessor of me in current
        send ANSWER(current, budget+cost(me, a_p), spent-cost(me, a_p), known,
         visited, k, c_i) to a_p
```

At starts, a_1 initializes the `known` *map with the paths to a_2 and a_3:*

`known = {` $a_1 \rightarrow a_2$ `: 1 ,` $a_1 \rightarrow a_4$ `: 1 }`

The first exploration of the graph (depicted in red) starts

- *a_1 starts by sending itself a* `REQUEST` *message with a budget of* 1.
- *When handling this message, a_1 forwards the request to a_2, with a budget of* 0.
- *Then a_2 fills out* `known` *and sends back an* `ANSWER` *to a_1, as the budget does not allow forwarding the request further.*
- *As it did with a_2, a_1 now sends a* `REQUEST` *to a_4, which fills out* `known` *and sends* `ANSWER` *back.*
- *At this point, a_1 as no other neighbor to forward the* `REQUEST` *and must increase the budget to* 2*, the cheapest path in* `known`.

The same process, depicted in green, is repeated with a budget of 2*:*

- *This updated budget allows expanding the path up to $\tilde{a}_2$, where a first replica is placed.*
- *A new path to a_4 is discovered but not kept in* `known`*, as it already contains a cheaper path to that node.*
- *Once all paths that can be reached with this budget have been explored, a_1 increases the budget again.*

Once again, a_1 explores the graph by `REQUESTS` *messages, this time with a budget of* 5 *(depicted in blue).*

- *This budget allows reaching $\tilde{a}_3$, and thus hosting the second replica.*
- `ANSWER` *messages are then sent back up to a_1, with* `k=0` *(all required replicas have been placed) and* ***DRPM*** *terminates.*

At the end of the process, replicas of c_i have been placed on a_2 and a_3, with path costs of respectively 2 *and* 5*. No replica has been placed on a_4, as it would incur a higher path cost of* 6.

Globally, each agent is responsible for placing k replicas for each of the active computations it currently hosts, and thus executes **DRPM** once for each of its active computations. These multiple **DRPM** runs can be either sequentially or concurrently executed, but their result depend on message reception order. Note however that even when running multiple **DRPM** concurrently, an agent has only one message queue and handle incoming messages sequentially, which prevents him from accepting replicas that would exceed its capacity.

Theorem 1. ***DRPM*** *terminates.*

Proof. For $k = 1$, since **DRPM** costs are additive and monotonic, and it bookkeeps paths to unvisited vertices, it terminates like classical iterative lengthening, with the minimum cost path or empty path if not enough memory in agents to host the computation x_i.

For $k > 1$, **DRPM** attempts to place each replica sequentially, it first searches for the best path (as for $k = 1$), then operates the same process for a second best path, and so on until either

1. the k replicas are placed (line 3 in Algorithm 2) or

2. there is not enough memory to host the n^{th} replica (line 16 in Algorithm 2).

Bookkeeping ensures the same path will not be considered twice, and thus consecutive search iterations output different paths with increasing path costs. So, in case (1), **DRPM** terminates when k replicas have been placed on the k best hosts; and in case (2), it terminates when $k' < k$ replicas have been placed, where k' is the maximum number of replicas that can be placed. □

5.5.4 Migrating Computations

Now that we have introduced **DRPM** to replicate computations, we can use these replicas to repair our system when an agent fails. We model the repair problem itself as a **DCOP**, to be implemented by agents to move some computations to restore the correct function of the system or to increase the quality of the distribution of the computations over agents.

Let's first introduce some notations.

We note $\mathcal{C}_c$ the set of candidate computations c_i that could or must be moved when the set of agents changes.

For each of these computations, we note $\mathcal{A}_c^i$ the set of candidate agents that could host c_i.

The set of all candidate agents, regardless of computations, is noted $\mathcal{A}_c$:

$$\mathcal{A}_c = \cup_{c_i \in \mathcal{C}_c} A_c^i$$

$\mathcal{C}_c^m$ denotes the set of candidate computations that agent a_m could host.

Deciding which agent $a_m \in \mathcal{A}_c$ hosts each computation $c_i \in \mathcal{C}_c$ can be mapped to an optimization problem similar to **ILP-CGDP** presented in Section 4.5.3, restricted to $\mathcal{A}_c$ and $\mathcal{C}_c$: communication and hosting costs should be minimized while honoring the capacity constraints of agents.

To ensure that each candidate computation is hosted on exactly one agent, we rewrite constraints (4.26) for each $c_i \in \mathcal{C}_c$:

$$\sum_{a_m \in \mathcal{A}_c^i} c_i^m = 1 \tag{5.10}$$

Similarly, capacity constraints (4.25) can be reformulated as:

$$\sum_{c_i \in \mathcal{C}_c^m} \mathbf{w}(c_i) \cdot c_i^m + \sum_{c_j \in \nu^{-1}(a_m) \setminus X_c} \mathbf{w}(c_j) \leq \mathbf{w_{max}}(a_m) \tag{5.11}$$

The hosting cost objective in (4.24) can be similarly formulated using one soft constraint for each candidate agent a_m:

$$\sum_{c_i \in \mathcal{C}_c^m} \mathbf{c_{host}}(a_m, c_i) \cdot c_i^m \tag{5.12}$$

Finally, the communication costs in (4.24) are represented with a set of soft constraints.

For an agent a_m, the communication cost incurred by hosting a computation c_i can be formulated as the sum of the cost of the cut edges (c_i, c_j) from the computation graph $\langle \mathcal{C}, D \rangle$, (i.e. where $\nu^{-1}(c_j) \neq a_m$).

Let's note N_i the neighbors of c_i in the computation graph. When a neighbor c_n is not a

candidate computation (i.e. it might not be moved and $c_m \in N_i \backslash \mathcal{C}_c$), the communication cost of the corresponding edge is simply given by $\mathbf{com_a}(c_i, c_j, a_m, \nu^{-1}(c_n))$.

For neighbors that might be moved, the communication cost depends on the candidate agent that is chosen to host it and can be written as $\sum_{a_n \in \mathcal{A}_c^j} c_j^n \cdot \mathbf{com}(i, j, m, n)$. With this we can write the communication cost soft constraint for agent a_m:

$$\sum_{(c_i,c_j) \in \mathcal{C}_c^m \times N_i \backslash \mathcal{C}_c} c_i^m \cdot \mathbf{com_a}(c_i, c_j, a_m, \nu^{-1}(c_j)) + \sum_{(c_i,c_j) \in \mathcal{C}_c^m \times N_i \cap \mathcal{C}_c} c_i^m \cdot \sum_{a_n \in \mathcal{A}_c^j} c_j^n \cdot \mathbf{com_a}(c_i, c_j, a_m, a_n) \tag{5.13}$$

We can now formulate the repair problem as a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ where

- $\mathcal{A}$ is the set of candidate agents A_c.
- $\mathcal{X}$ and $\mathcal{D}$ are respectively the set of decision variables c_i^m and their domain $\{0, 1\}$.
- $\mathcal{C}$ is composed of constraints (5.10), (5.11), (5.12), and (5.13) applied for each agent $a_m \in A_c$. (5.10) and (5.11) result in infinite costs when violated, while (5.12) and (5.13) directly define costs to be minimized.
- The mapping function μ assigns each variable x_i^m to agent a_m.

Definition 22 (DMCM)**.** *Given a set of candidate computations $\mathcal{C}_c$ and a set of candidate agents $\mathcal{A}_c$, we term **DCOP Model for Computation Migration (DMCM)** the **DCOP** model for selecting a suitable agent for each of the computations.*

Example 18. *Figure 5.5 represents a factor graph for the **DCOP** modeling the migration decision for a computation c_i, which could be placed on two agents a_1 and a_2.*

Hard constraints are depicted in red.

- *capa$_1$ and capa$_2$ represent constraints 5.11 for a_1 and a_2 and ensure that the capacity of these agents is not exceeded.*
- *hosted$_i$ represents constraint 5.10 and ensures that exactly one agent hosts c_i.*

Soft constraints are represented in green.

- *hosting$_1$ and hosting$_2$ represent constraints 5.12 for a_1 and a_2 and minimize the hosting costs.*
- *comm$_i$ represents constraint 5.13 and minimizes the communication costs.*

Notice that this model for computations migration is not specifically designed for fixing the system after agents failure; it simply implements the decision process for selecting a suitable agent to host some computation(s). As a consequence, it can be used both to implement repair, which we will present in the next section (5.5.5), and to re-distribute computations after some agent(s) arrival, which will be discussed in Section 5.6.3.

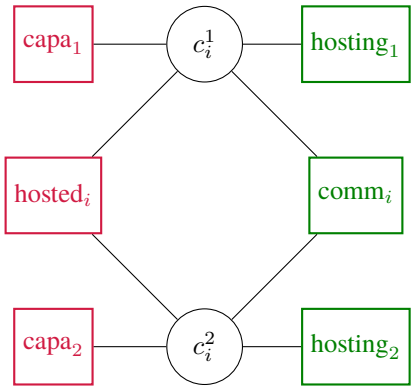


Figure 5.5 – Factor graph representation of a the **DCOP** model for migrating computation c_i

5.5.5 Implementing Repair using DRPM[DMCM]

Using the **DCOP**-based model for selecting an agent when migrating a computation, we can now implement the repair phase of k-resilience. Indeed, when up to k agents fail, repairing the system amounts to migrate each of the orphaned computations to one of the agents that possess its replica.

Definition 23 (DRPM[DMCM])**.** *We term **DRPM[DMCM]** the full solution method for k-resilience composed of **DRPM** for replication and the **DMCM** model for computation migration.*

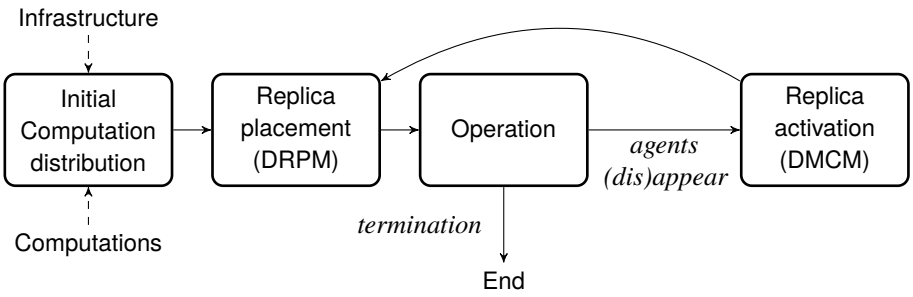


Figure 5.6 – **DRPM[DMCM]** life cycle in a glance.

Figure 5.6 represents the life cycle of this approach. Assuming initial deployment (using one of the methods discussed in Section 4) and replicas placement (using **DRPM**) have been performed at system bootstrap, the system will execute the following repair cycle all along its lifetime:

(a) Detect departure/arrival,

(b) Activate replicas of missing computations, by solving the **DCOP** for computation migration,

(c) Place new replicas for missing computations using **DRPM**, and continue nominal operation.

As discussed in Section 5.3.2.1, step (a) assumes some discovery and *keep alive* mechanisms that automatically inform some agents of any events in the infrastructure. So when an agent a_m fails or is removed, we consider that all neighbor agents of a_m in the **route**–graph are aware of the departure.

Step (b) relocates computations that were hosted on the set of departed agents $\mathcal{A}_d$ to other agents. The candidate computations $\mathcal{C}_c$ are the orphaned computations hosted on these agents:

$$\mathcal{C}_c = \cup_{a_m \in \mathcal{A}_d} \nu(a_m)$$

To avoid extra delay and communication during the repair phase these orphaned computations should be assigned to agents that already have the necessary information to run an active version of the computation. This means that the set of candidate agents $\mathcal{A}_c$ for an orphaned computation c_i maps the set of still available agents hosting a replica for this computation:

$$\mathcal{A}_c^i = \rho(c_i) \backslash \mathcal{A}_d$$

In a k-resilient system, as long as $|\mathcal{A}_d| \leq k$, we are sure that there will always be at least one agent in $\mathcal{A}_c^i$. Thus, step (b) yields an assignment of each of the orphaned computations to one of the remaining agents hosting its replica.

Step (c) maintains a good resilience level in the system by repairing the replica distribution using **DRPM** on a smaller problem, since many replicas are already placed.

5.5.6 Solving DMCM using a DCOP Algorithm

Now that the repair problem has been expressed as a **DCOP**, we discuss its resolution using a DCOP solution method. Many solution methods for DCOPs exist, several of which have be presented in Section 2.4. In brief, using these message passing protocols, agents coordinate to assign values to their variables. Each of these solution methods has specific characteristics (they might be complete or not, synchronous or asynchronous, etc.) and makes some assumptions on the environment and the problem (perfect message delivery, hard and/or soft constraints, etc.). Therefore, when solving a **DCOP**, it is very important to select a **DCOP** algorithm that matches the characteristics of the problem and its environment.

In the case of **DMCM**, we can identify the following key characteristics to guide our choice of suitable solution methods:

(a) The problem must be solved by constrained devices.

(b) A solution must be found as quickly as possible, as the system will only get back to nominal operation once all orphaned computations have been successfully migrated.

(c) Our model contains both hard constraints, to ensure all orphaned computation are migrated and that agent's capacity is honored, and soft constraints, optimizing for hosting and communication costs.

(d) A suboptimal solution is acceptable, as long as all hard constraints are satisfied. Indeed, we can reasonably sacrifice some optimality on hosting and communication cost, if orphaned computations are migrated to agents that have enough capacity to host them. Additionally, violating a hard constraint could also mean loosing an orphaned computation, or activating several replicas for the same computation. In both cases, the system will be in an inconsistent state.

(e) As **DMCM** is designed to repair a distribution, the solution method used to solve it must not bring about a distribution problem itself, otherwise we would have a *chicken and egg* situation . . .

Characteristics (a), (b) and (c) compel us to select a lightweight suboptimal algorithm. Local

search algorithms for instance, are fast and require very little computation on each agent.

Characteristic (e) implies that we must use an algorithm for which the assignment of computations to agents is fully defined for the **DMCM** problem. As this model contains only binary variables c_i^m, where m maps to agent a_m, we can easily map each variable to an agent. As a consequence, by using constraint graph-based algorithms, which only define computations for variables, me make sure we do not have a distribution problem when deploying the **DCOP** used to solve **DMCM**.

Characteristic (c) is more difficult to satisfy. As a matter of fact, few **DCOP** algorithms have been designed specifically to take into account a mix of hard and soft constraints and many iterative algorithms tend to break hard constraints when optimizing for soft constraints. A monotonic algorithm, like **MGM** [77] (see Section 2.4.2.2 for a full description) is particularly well suited for this situation; as the cost of the solution monotonically decreases, once the hard constraints (modeled with infinite costs) have been satisfied they will not be broken while optimizing the soft constraints. In our case, decisions require coordination between two agents: to move a computation from agent a_m to agent a_p, the binary variable c_i^m must take 0 as a value, while *simultaneously*, c_i^p must switch from 0 to 1. This need for simultaneous changes justifies the use of **MGM-2** [77] (**Maximum Gain Message with 2-coordination**).

By applying **MGM-2** to **DRPM[DMCM]**we obtain a repair method that we term **DRPM[MGM-2]**. Of course, we could use any other **DCOP** algorithm that matches the characteristics (a), (b), (c), (d) and (e) identified previously.

5.6 Handling Agent Arrival

Now that we have seen several approaches for dealing with agent's departure, we will see how they could be extended to also handle the arrival of a new agent.

5.6.1 In the Neighborhood

In Section 5.4 we introduced an approach for dealing with an agent departure by solving **ILP-CGDP** in a sub-graph defined by the neighborhood of the agent. We will now see that the same approach could also be used when a new agent enters the system. As a matter of fact, this approach only requires us to define a neighborhood of an agent.

As stated in Section 5.3.1.2, when a new agent enters the system, it can either be a *participating agent*, which already host some computation(s), or be *blank*, in which case it only provides computation and memory, without hosting any computation. In order to benefit from these capabilities, existing computations may be relocated to the newcomer. In this case, the re-deployment process amounts to selecting the elements to migrate as to optimize communication costs.

In the case of a participating agent, no change to our initial approach is needed: we can straightforwardly apply the neighborhood Definition (Definition 17) and solve **Integer Linear program for CGDP restricted to the neighborhood (ILP-CGDP**$[a_k]^+$**)**, the cut version of **ILP-CGDP** (Definition 15) restricted to the sub-graph defined by this neighborhood.

Definition 24 (ILP-CGDP$[a_k]^+$)**.** ***ILP-CGDP***$[a_k]^+$ *consists in* ***ILP-CGDP*** *(Definition 15) restricted to the set of agents* $\mathcal{A}[a_k]$ *and to the computation graph* $\langle \mathcal{C}[a_k], E[a_k] \rangle$.

The case of a blank agent is more complicated according to Definition 17, its neighborhood is the entire agent system. That means that we would need to solve **ILP-CGDP** over the whole computation graph, which is not feasible on our constrained devices. As a consequence, in order to apply this approach, we would need to devise a method to select a subset of agents as a neighborhood to solve **ILP-CGDP** on. One could select agents that have the lowest remaining capacity, for example, or agents that are more prone to failure or disconnection in order to improve the resilience of the system by hosting as much computation as possible on stable agents.

ILP-CGDP$[a_k]^+$ can be in both cases solved using the techniques listed in Section 5.4.3. Of course, this approach suffers form the same limitations as identified in 5.4.4.

On top of these limitations, we consider that the main challenge when handling device arrival with **ILP-CGDP** lies in the definition of the neighborhood for the blank agents.

5.6.2 Newcomer Decision Problem for Agent Arrival

As we have seen, re-using the previously defined **ILP-CGDP** to handle device arrival is possible but requires to define a neighborhood for the newcomer agent, which may be difficult, and suffers from the high computation and communication loads incurred by distributed approaches for solving it.

Here, as to avoid these high loads induced by the previous technique, we consider a more newcomer-centric approach: the newcomer agent calls for proposals to move some computations; then, based on the costs of the proposed computations and its own memory capacity, the newcomer has to choose a set of computations to host.

Another benefit of this approach is that it delegates the definition of the subset of the computation graph (which is equivalent to the definition of the neighborhood in the previous technique) to the agents that are already active in the system, and should thus have a better knowledge to make the right decision.

Let's formulate this newcomer decision problem.

Definition 25 (CGDP-NDP)**.** *Given a newcoming agent and a set of proposed computations to migrate coming from its neighborhood, the **Newcomer Decision Problem for the Computation Graph Distribution Problem (CGDP-NDP)** amounts to choose computations amongst proposed computations, so that communication load and hosting costs are minimized and memory constraints are fulfilled.*

Practically, when a new agent a_k enters the system, the distributed discovery mechanism (see 5.3.2.1) informs other agents of that arrival, which would typically be implemented by some kinds of broadcast communication. This type of communication is unreliable, meaning that the set of agents receiving this information is at most the whole set of agents $\mathcal{A}$ but more generally a subset of $\mathcal{A}$ defined by network proximity and the discovery mechanism used in the system.

In the following, we note $\mathcal{A}[a_k]^D$ the subset of agents informed of the arrival of the newcomer by the discovery mechanism. If the newcomer is a participating agent, this subset includes at least the agents that host a computation connected to a computation hosted on a_k –that is to say, it is a least the neighborhood $\mathcal{A}[a_k]$, as defined in (Definition 17).

For the communication costs, we reuse $\mathbf{com}(c_i, c_j)$, as defined in Section 4.4. Indeed, as we will see shortly, the problem cannot be efficiently solved when using the more general definition of communication costs, $\mathbf{com_a}(c_i, c_j, a_m, a_n)$ defined in Section 4.5.2.

Each agent $a_\ell \in \mathcal{A}[a_k]^D$ sends its proposal to a_k in a message made of a tuple $\langle \mathcal{C}^{\ell\to k}, E^{\ell\to k}, \mathbf{com}, \mathbf{host}\rangle$, where:

- $\mathcal{C}^{\ell\to k} \subset c$ is the set of computations hosted on a_ℓ and proposed to be migrated to the newcomer a_k,
- $E^{\ell\to k} = \{(c_i, c_j) \mid (c_i, c_j) \in E, c_i \in \mathcal{C}^{\ell\to k} \text{ or } e_j \in \mathcal{C}^{\ell\to k}\}$ is the set of edges connected to computations in $\mathcal{C}^{\ell\to k}$,
- $\mathbf{com}$ is the communication cost function (potentially restricted to elements in $E^{\ell\to k}$),
- $\mathbf{host}$ is the hosting cost of each proposed computation on its current agent.

We note $\mathcal{C}^k$ the set of computations proposed to the newcomer agent:

$$\mathcal{C}^k = \bigcup_{a_\ell \in \mathcal{A}[a_k]^D} \mathcal{C}^{\ell\to k}$$

and E^k the set of edges derived from these proposals:

$$E^k = \bigcup_{a_\ell \in \mathcal{A}[a_k]^D} E^{\ell\to k}$$

Notice that E^k may contain edges involving computations that are not in $\mathcal{C}^k$, and thus cannot be migrated, as we must also account in our decision for communication costs with these computations. We denote $\mathcal{C}^{k+}$ the set of computations connected to at least one edge in E^k, even the ones that are not movable (thus, not necessarily proposed for migration):

$$c^{k+} = \{c_i \mid (c_i, c_j) \in E^k \text{ or } (c_j, c_i) \in E^k\}$$

We assume the communication cost $\mathbf{com}(c_i, c_j)$ can be assessed only using information sent by proposers.

Example 19. *Figure 5.7 represents the proposals from agents a_1 and a_2 to newcoming agent a_k. Agent a_1's proposal, depicted in yellow, includes c_1 and thus:*

$$\mathcal{C}^{1\to k} = \{c_1\} \text{ and } E^{1\to k} = \{(c_1, c_2), (c_1, c_3), (c_1, c_5)\}$$

Similarly for a_2's proposal, depicted in blue includes c_3:

$$\mathcal{C}^{2\to k} = \{c_3\} \text{ and } E^{2\to k} = \{(c_3, c_5), (c_1, c_3)\}$$

The set of computations proposed to a_k is $\mathcal{C}^k = \{c_1, c_3\}$ and the set of associated edges is:

$$E^k = \{(c_1, c_2), (c_1, c_3), (c_1, c_5), (c_3, c_5)\}$$

Notice that some edges in E^k involve a computation that is not proposed, like for instance here c_5 and c_2.

$$\mathcal{C}^{k+} = \{c_1, c_2, c_3, c_5\}$$

Each proposal also includes the communication cost for each of the edges in $E^{\ell \to k}$. and the hosting cost of each proposed computation on its current agent.

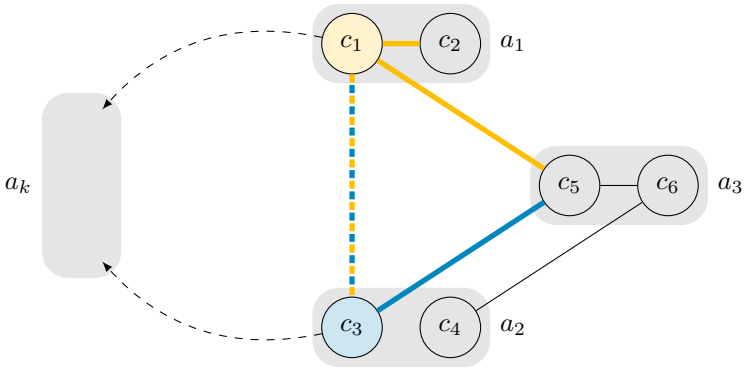


Figure 5.7 – Sample proposals from agents a_1 and a_2 to newcoming agent a_k

Let e_i^k be a binary variable stating whether the newcomer a_k chooses to host computation c_i.

The cost of selecting a set of computations can be formulated as the sum of:

(a) the communication cost of edges that are cut (i.e. the ends or the edge are hosted on different agents) by moving the selected computation to a_k,

(b) the negative communication costs of edges whose ends are now both hosted a_k,

(c) the difference of hosting costs, for each selected computation, between the hosting cost on their current agent and a_k.

Case (a) maps to edges $(c_i, c_j) \in E^k$ for which e_i^k `XOR` e_j^k holds true (i.e. exactly one of the computations is selected by a_k). We can reformulate the `XOR` operator with binary variables as follow: $e_i^k + e_j^k - 2 \cdot e_i^k \cdot e_j^k$.

Case (b) maps to edges that were cut before the selection and would whose both ends are now hosted on a_k, i.e. those for which e_i^k `AND` e_j^k holds true, which can be rewritten as $e_i^k \cdot e_j^k$.

Case (c), for each selected computation c_i, is simply the difference in hosting costs: $\mathbf{host}(\nu(c_i), c_i) - \mathbf{host}(a_k, c_i)$.

By summing these two components, we can write the impact of on communication cost of selecting a set of computations as follows:

$$\begin{aligned} &\sum_{(c_i,c_j)\in E^k} \mathbf{com}(c_i, c_j)(e_i^k + e_j^k - 2 \cdot e_i^k \cdot e_j^k) \\ -&\sum_{(c_i,c_j)\in E^k} \mathbf{com}(c_i, c_j) \cdot e_i^k \cdot e_j^k \end{aligned} \tag{5.14}$$

And the impact on hosting cost can be expressed as:

$$\sum_{c_i \in \mathcal{C}^k} e_i^k \cdot (\mathbf{host}(\nu(c_i), c_i) - \mathbf{host}(a_k, c_i)) \tag{5.15}$$

Example 20. *When accepting c_1 and c_3 (Figure 5.8):*

- *Edge (c_1, c_2), depicted in red, was previously entirely contained in a_1 and was thus not incurring any communication cost. This edge is now cut and we must account for the cost $\mathbf{com}(c_1, c_2)$.*
- *Edge (c_1, c_3), depicted in green, was previously cut between agents a_1 and a_2. This edge is now entirely contained in a_k and thus we do not have the corresponding cost any more: $\mathbf{com}(c_1, c_3)$.*
- *We now have to account for the costs for hosting c_1 and c_3 on a_k*

Thus, the overall cost of selecting c_1 and c_1 is equal to:

$$\begin{aligned}&\mathbf{com}(c_1, c_2) - \mathbf{com}(c_1, c_3)\\ &+ \mathbf{host}(a_1, c_1) - \mathbf{host}(a_k, c_1) + \mathbf{host}(a_2, c_3) - \mathbf{host}(a_k, c_3)\end{aligned}$$

Notice that communication costs do not change for edges (c_1, c_5) and (c_3, c_5) as the definition of communication used here does not take agents into account.

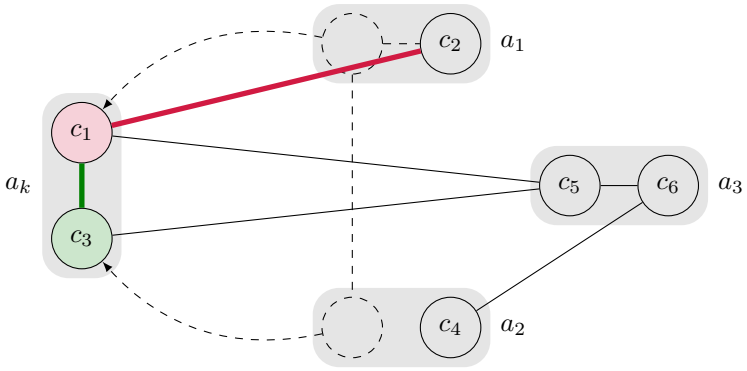


Figure 5.8 – Communication costs when accepting proposals from a_1 and a_2

We can sum these two components (with some simplification) and use the result as the optimization objective for the newcomer a_k, as follows:

$$\underset{e_i^k, e_j^k}{\textbf{minimize}} \sum_{(c_i,c_j) \in E^k} \mathbf{com}(c_i, c_j)(e_i^k + e_j^k - 3 \cdot e_i^k \cdot e_j^k) + \sum_{c_i \in \mathcal{C}^k} e_i^k \cdot (\mathbf{host}(\nu(c_i), c_i) - \mathbf{host}(a_k, c_i)) \tag{5.16}$$

subject to

$$\sum_{e_i \in V^k} \mathbf{mem}(e_i) \cdot e_i^k \leq \mathbf{w_{max}}(a_k) \tag{5.17}$$

Definition 26 (IQP-CGDP-NDP)**.** *We term Integer Quadratic Problem for CGDP-NDP (IQP-CGDP-NDP) the* $0/1$ *integer quadratic program consisting of quadratic objective (5.16) and linear constraints (5.17) which encodes CGDP-NDP (Definition 25).*

We will now show that this problem falls into the **Quadratic Knapsack Problem (QKP)** framework.

The communication part of Equation (5.14) can be reformulated as follows:

$$\begin{aligned}
&\sum_{(c_i,c_j)\in E^k} \mathbf{com}(c_i,c_j)(e_i^k + e_j^k - 3\cdot e_i^k \cdot e_j^k) \\
&= \sum_{(c_i,c_j)\in E^k} \mathbf{com}(c_i,c_j)(e_i^k + e_j^k) + \sum_{(c_i,c_j)\in E^k} -3\cdot \mathbf{com}(c_i,c_j) e_i^k \cdot e_j^k \\
&= \sum_{c_i\in\mathcal{C}^k} e_i^k \cdot \sum_{c_j\in\mathcal{C}^{k+}} \mathbf{com}(c_i,c_j) + \sum_{(c_i,c_j)\in E^k} -3\cdot \mathbf{com}(c_i,c_j) e_i^k \cdot e_j^k \\
&= \sum_{e_i\in\mathcal{C}^k} e_i^k \cdot \mathbf{p}(c_i) + \sum_{c_i\in\mathcal{C}^k} \sum_{c_j\in\mathcal{C}^k, i\neq j} e_i^k \cdot e_j^k \cdot \mathbf{P}(c_i,c_j)
\end{aligned}$$

with

$$\mathbf{p}(c_i) = \sum_{c_j\in V^{k+}} \mathbf{com}(c_i,c_j), \quad \forall e_i \in V^k$$

$$\mathbf{P}(c_i,c_j) = \begin{cases} -3\cdot\mathbf{com}(c_i,c_j), & \text{if } (c_i,c_j)\in E^k \\ 0, & \text{otherwise} \end{cases}$$

We can now add the hosting costs (from Equation 5.15) and rewrite the objective as follows:

$$\begin{aligned}
&\sum_{e_i\in\mathcal{C}^k} e_i^k \cdot \mathbf{p}(c_i) \sum_{c_i\in\mathcal{C}^k} \sum_{c_j\in\mathcal{C}^k, i\neq j} e_i^k \cdot e_j^k \cdot \mathbf{P}(c_i,c_j) + \sum_{c_i\in\mathcal{C}^k} e_i^k \cdot (\mathbf{host}(\nu(c_i),c_i) - \mathbf{host}(a_k,c_i)) \\
&= \sum_{e_i\in\mathcal{C}^k} e_i^k \cdot \mathbf{p}'(c_i) \sum_{c_i\in\mathcal{C}^k} \sum_{c_j\in\mathcal{C}^k, i\neq j} e_i^k \cdot e_j^k \cdot \mathbf{P}(c_i,c_j) \qquad (5.18)
\end{aligned}$$

with

$$\mathbf{p}'(c_i) = \mathbf{host}(\nu(c_i),c_i) - \mathbf{host}(a_k,c_i) + \mathbf{p}(c_i)$$

With our objective rewritten as Equation 5.18, which is the canonical form of **QKP**, we can see that the **IQP-CGDP-NDP** falls into the **QKP** framework. Notice that this assumes that $\mathbf{w_{max}}(a_k)$ are $\mathbf{w}(c_i)$ are positive and integral, which is reasonable in our case, as they represent respectively the capacity of an agent and the footprint of a computation (see Section 4.4).

QKP can be linearized [3] and then we could solve it using a centralized branch-and-cut method or a distributed optimization method, as discussed in Section 5.4.3.

More interestingly in our case, **QKP** is solvable using a dynamic programming heuristic [41], but without optimality guarantees. Only requiring $\mathcal{O}(\mathbf{w_{max}}(a_k).|c^k|)$ space, and $\mathcal{O}(\mathbf{w_{max}}(a_k).|c^k|^2)$ time, such a lightweight heuristic approach seems realistic in our case.

Besides, instead of using its whole memory capacity $\mathbf{w_{max}}(a_k)$, device a_k may also set a limit capacity below its maximum one (e.g. the average memory used by its neighbors) as not to host more computation than others, in general.

In case the number of proposals is too high, a_k may also choose to only consider a randomly chosen set of proposers.

Using the heuristic and these two approaches (reducing available capacity and limiting the number of proposed computation), the problem can be made very easy, and can be solved in the newcoming agent, even though it is a constrained device.

From the already active agents side, the decision of making a proposition, and of choosing which computations to propose, is also an issue. The easiest solution is to propose all "movable" computations, i.e. computations like shared decision, that are not tied to an agent by the specific problem's domain characteristics. In the case of **SECP**, that would be for instance computation representing physical models and rules, while actuator computations would never be proposed.

Of course, one could also use more elaborate strategies, where agents would make a proposal only when they actually need it, for instance when they are already hosting many computations and have limited remaining capacity. This would allow exploiting the knowledge of the system that these active agents have, something that was not possible in the previous approach.

As a conclusion, this approach is lightweight and has very interesting characteristics compared to the previously introduced **ILP-CGDP**$[a_k]^+$. However, it is restricted to a simplified definition of the communication costs and cannot be used with the more general definition $\mathbf{com_a}$ (see Section 4.5.2), as, when using the $\mathbf{route}$ costs between agents, the problem cannot to be cast into a **QKP** and thus we cannot use the efficient heuristic from [41].

5.6.3 DMCM-based Approach for Agent Arrival

In Section 5.6.1, we stated that **ILP-CGDP**$[a_k]^-$, the approach we introduced in Section 5.4 to deal with agent departure, could also be re-used when a new agent enters the system. The only issue is to select a subset of agents to be considered when restricting **ILP-CGDP**.

As a matter of fact, this is also true for the **DMCM** repair method introduced in Section 5.5.4. All we need to use the **DMCM** model is to define a set of candidate agents and candidate computations; once these sets are defined, **DMCM** can be applied to agent arrival, and solved in a distributed manner using a **DCOP**. As discussed in Section 5.6.1, the appropriate definitions for these sets depend on the problem and system's characteristics.

Notice that when dealing with agent's arrival, we do not need **DRPM**, the replication component of our k-resilience framework **DRPM**[**DMCM**]. As computations are migrated from active agents to the newcoming agent, their definition is still available and can simply be transmitted once the migration has been agreed on.

5.7 Experimental Evaluation

In the next sections, we experimentally analyze the performances and behaviors of our different contributed algorithms to handle single-agent arrival and departure (Section 5.7.1), to replicate

computation definitions (Section 5.7.2), to repair systems where agents disappeared (Section 5.7.3) and to install resilience (Section 5.7.4).

5.7.1 Handling Agent Arrival and Departure

In this first set of experiments, we evaluate the performances of **ILP-CGDP**$[a_k]^-$, **ILP-CGDP**$[a_k]^+$ and **CGDP-NDP**, our solution methods for migrating computations when a single agent joins or leaves the system.

These experiments are performed on **SECP** problems, solved using **MaxSum**, as we stated in Section 3.3 that this family of algorithms was well suited for the characteristics of these problems.

In our simulations, two types of events may occur: device arrival (in) and unsafe device removal (out).

- In case of device arrival, we use either **ILP-CGDP**$[a_k]^+$ (Section 5.6.1), which is solved using a classical ILP solver within one node (using GLPK in our simulator), or **CGDP-NDP** (Section 5.6.2), which is solved using a dedicated dynamic program (embedded in our simulator, in Python). Each new device arriving in the system represents a *participating agent* (Section 5.3.1.2): it is already connected to models and rules in the **SECP** and its neighborhood is thus well defined.
- In case of device removal **ILP-CGDP**$[a_k]^-$ is solved using GLPK. Notice that we do not consider in this experiment the issue the availability of computations' definition, discussed in Section 5.4.4. We simply assume here that, when repairing, any agent in the neighborhood can access the computations' definition.

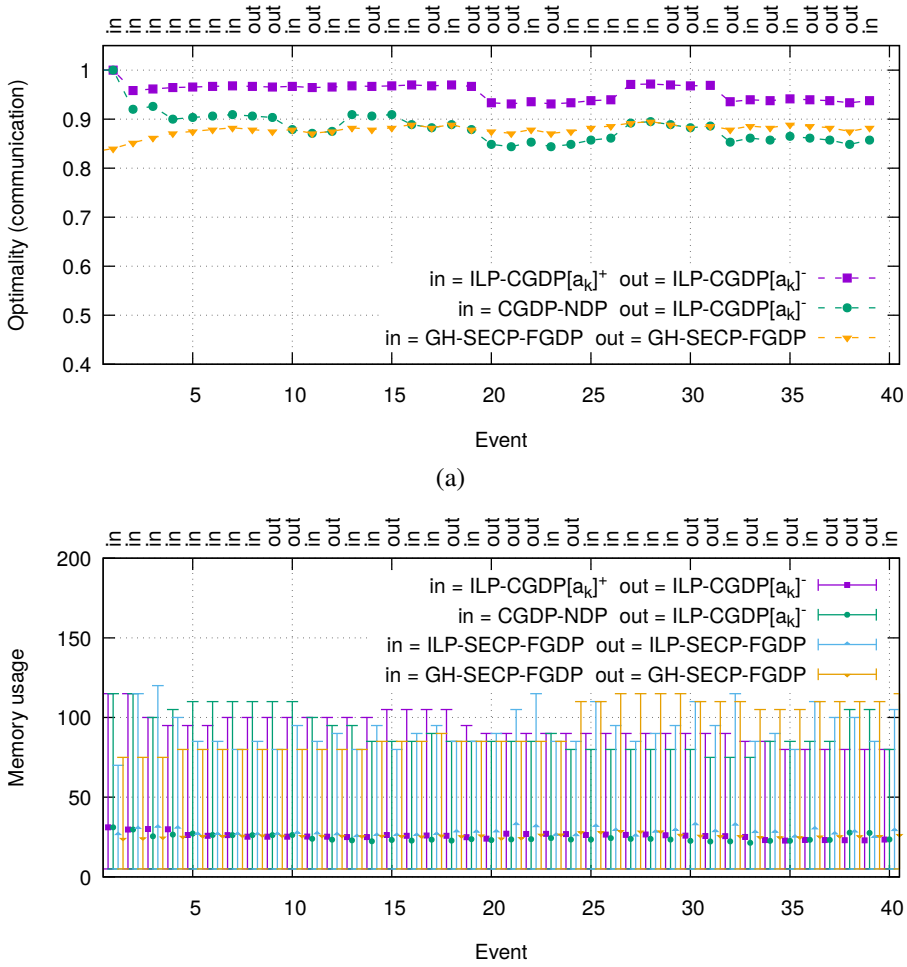
Notice that our distribution here is based on the **SECP**-specific definition from Section 4.4, which allows use to compute an optimal distribution for relatively large problems. Besides, **CGDP-NDP** is only defined for distribution methods based on the communication costs, like **ILP-SECP-CGDP**.

Whatever the type of event, the best **ILP-SECP-FGDP** solution (computed with GLPK), and the solution provided by the **GH-SECP-FGDP** heuristic are computed to benchmark aforementioned methods.

Notice also the discrepancies in terms of solution method implementation (GLPK vs python code): for this reason, we do not plot the repair times (which always take at most a few seconds). Instead, we concentrate on the evaluation of the quality of the distribution after repair.

5.7.1.1 Simulated Smart Home Scenarios

In a first series of experiments, we simulate the first floor of a real smart home, as represented in Figure 3.7 (presented in Section 3.2.2 on page 46), which is initially composed of 13 actuators (light bulbs and their respective costs), 6 physical models (one for each space), and 5 user rules (not represented in the figure, for clarity). The default agent memory capacity ($\mathbf{w_{max}}$) is set to 200 memory units (one unit represents the space to store one value, e.g. 32 bits). Figure 5.9 traces performances of repair solutions on a scripted scenario where devices are added and removed at runtime. Each of these 40 events is followed by a repair phase using the proposed methods.



(a)

(b)

Figure 5.9 – Optimality (5.9a), and memory usage (5.9b) of the deployment during the simulation (standard deviation, min and max)

Figure 5.9a shows the quality of the distribution after handling an in or out event, computed as the ratio between the repaired distribution cost and the best cost (real **ILP-SECP-FGDP** optimum). The centralized **GH-SECP-FGDP** heuristic is also plotted for comparison. Clearly, with both approaches, out events tends to degrade the optimality of the deployment, while still maintaining it at a very competitive level, compared to a full deployment of the whole factor graph. Interestingly, in events improve optimality, meaning that in real systems where on average out are approximately balanced by in, the deployment should keep a very good quality level.

As we also add devices in the system, it's interesting to see if we benefit from the newly added capacities and if the computations are evenly spread across the devices. Figure 5.9b presents the standard deviation (and min and max) of memory usage over all the devices, after each event. For comparison, it also includes the values obtained with and optimal distribution (with **ILP-SECP-FGDP**) and the **GH-SECP-FGDP**. While our approaches are not specifically designed to ensure a fair memory load share among devices, both distributed methods do not lead to an excessive accumulation of computations on a single device and perform at least as well as the two centralized approaches. Solving **ILP-CGDP**$[a_k]^+$ is a better choice in this regard, which can be explained by the fact that it allows relocation of computations on the full neighborhood, while

solving **CGDP-NDP** only allows migration of computations to the newcomer.

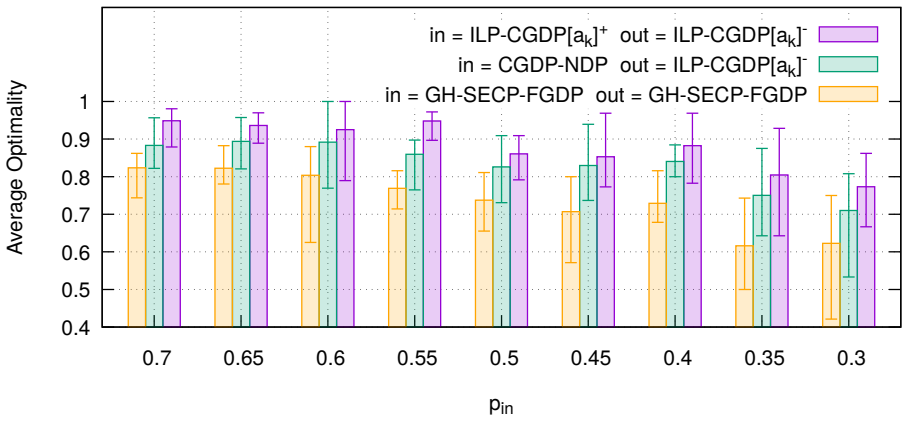
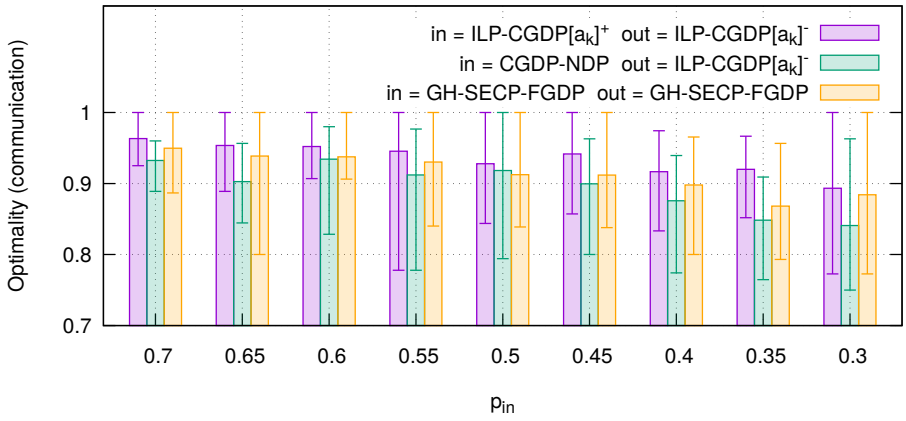


Figure 5.10 – Influence of the p_{in} probability on the distribution quality

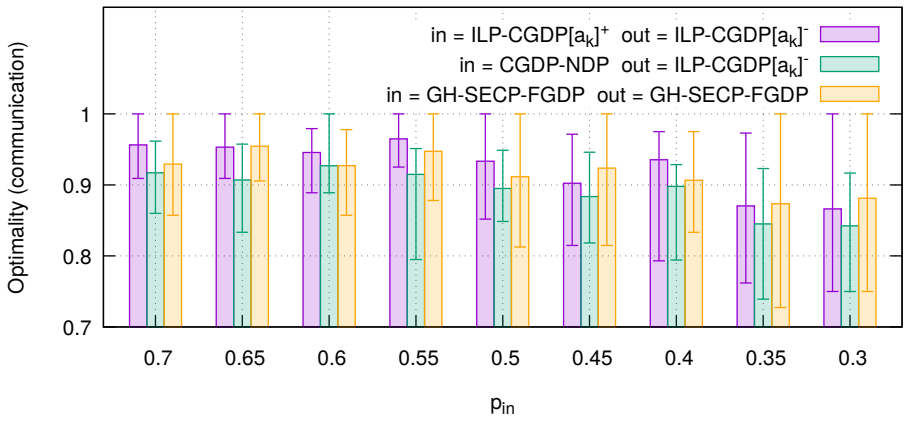
In a second series of experiments, we simulate the whole house with 23 actuators, 9 physical models and 9 rules. Here we evaluate the robustness of each repair techniques with more and more device removal. Figure 5.10 shows the average performances over 10 simulations after 20 events, in terms of distribution quality (computed as previously) with a varying event type probability. At each event generation, its type is determined using p_{in}, i.e. the probability for an event to be in. The higher p_{in}, the easier the adaptation is, since more devices are probably added. **ILP-CGDP**$[a_k]^+$ combined with **ILP-CGDP**$[a_k]^-$ presents very good resilience, since it offers more than 80% optimality with $p_{\text{in}} \geq 0.35$ (approximatively 2 removals for 1 arrival). **CGDP-NDP** combined with **ILP-CGDP**$[a_k]^-$ is always 5 to 15% lower. It is remarkable that these local repair techniques yield better distributions, from a communication point of view, than **GH-SECP-FGDP**, even though it is a centralized approach and has access to information about the whole factor graph. Finally, **ILP-CGDP**$[a_k]^+$ presents better optimality, but requires much more information to be computed, whilst **CGDP-NDP** is in average 10% worse in communication cost.

5.7.1.2 Randomly Generated SECPs

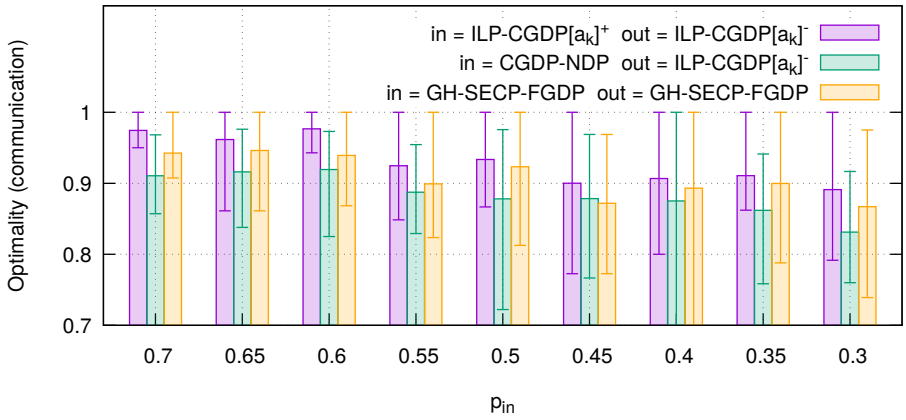
In a third series of experiments, we evaluate the influence of the number of rules in the **SECP** on the performance of the repair techniques. Here we generate 10 pairs of **SECP** and scenarios (containing 20 events) for each combination of p_{in} and n_r where $0.3 \leq p_{\text{in}} \leq 0.7$ and $10 \leq n_r \leq 50$ is the number of rules (with a step of 10). All **SECP** are generated randomly with 30 lights and 7 models and map to connected factor graphs, meaning that an increase on the number of rules also results in an increase on the factor graph density.



(a) **SECP** with 10 rules



(b) **SECP** with 30 rules



(c) **SECP** with 50 rules

Figure 5.11 – Influence of the p_{in} probability on the optimality for SECP with an increasing number of rules

Figure 5.11 shows the average performance in term of distribution quality (i.e. communication optimality). We can see that the good resilience of the local distribution repair approaches is not really impacted by the number of rules in the system ; results are very similar to those of the second experiment for both **ILP-CGDP**$[a_k]^+$ combined with **ILP-CGDP**$[a_k]^+$ and **CGDP-NDP** combined with **ILP-CGDP**$[a_k]^-$. However, we notice that the **GH-SECP-FGDP** heuristic performs much better than on the second experiment and consistently returns better distribution than **CGDP-NDP** combined with **ILP-CGDP**$[a_k]^-$. This can be explained by the fact that the

SECP used here are generated randomly while the **SECP** used for previous experiments were modeling actual real smart homes. Real **SECP** tends to have some locally semi-independent subgraphs, which roughly maps the various rooms and zones of a house. This structure is not present is random **SECP**, which tends to be much more uniform. This exhibits the high impact of the topology of the factor graph on the efficiency of the distribution approach. It is remarkable that the two local approaches are not much impacted by this change in topology.

5.7.2 Replication

In this section, we evaluate **DRPM**, our replicas placement algorithm, on several problem types and several multi-agent infrastructures.

5.7.2.1 Evaluation of DRPM on Benchmark Problems

First, we evaluate **DRPM** on standard **DCOP** benchmark problems: graph coloring problems on random and scale free graph.

Random soft coloring problems are generated by creating a random graph with density $p = 0.3$. Each edge is mapped to a binary constraint and each vertex is mapped to a variable with a domain made of 5 colors. Scale free problems are derived, using the same approach, from scale free graphs generated using the Barabàsi-Albert model [8] (starting from a 2-node connected graph), which are known to adequately model **IoT** systems [151]. In both cases, we generate instances with an increasing number of variables: $|\mathcal{X}| \in \{10, 20, \dots, 90\}$. For each variables count, we generate 10 instances and derive a computation graph for each of these instances, for algorithms based on a constraint graph representation.

As the distribution of replicas does not only depends on the problems definition, but also on the characteristics of the multi-agent infrastructure (agents and communication among them) used to solve it, we also generate two different infrastructures for each graph coloring problem: an uniform infrastructure and a problem-dependent infrastructure. An infrastructure is made of $|\mathcal{A}|$ agents, each holding one decision variable ($|\mathcal{A}| = |\mathcal{X}|$), and is defined by $\mathbf{cost}$, $\mathbf{route}$, and $\mathbf{w_{max}}$ (see Section 4.5 on distribution for the definitions) as follows.

The **uniform infrastructure** considers systems where communication costs between agents are uniform: $\forall a_m, a_n, \mathbf{route}(a_m, a_n) = 1$.

In the **problem-dependent infrastructure**, route costs $\mathbf{route}(a_m, a_n)$ are defined in a way that respects the structure of the computation graph: agents with many neighbors have a low communication cost while agents few neighbors have an higher communication cost. The idea is to model the structure found in many physical infrastructures like IoT, where powerful servers are connected to many other servers through high-performance networks, while small connected devices are using constrained connections.

More precisely, $\mathbf{route}(a_m, a_n) = \frac{1+||N(a_m)|-|N(a_n)||}{|N(a_m)|+|N(a_n)|}$ where $|N(a_m)|$ is the number of neighbors of a_m in the computation graph.

Additionally, we define hosting costs, agents' capacities and computations footprints and communication loads for both infrastructure and all problems instances as follows:

(i) hosting costs $\mathbf{cost}(a_m, x_j) = 0$ if the computation c_j is responsible for the variable initially assigned to agent a_m, $\mathbf{cost}(a_m, c_j) = 10$ otherwise;

(i) the capacity of each agent depends on the weight of its decision variable and is set to a large value, to ensure that all replicas can be hosted and k-resiliency is possible, even after several repairs: $\mathbf{w_{max}}(a_i) = 100 * \mathbf{mem}(c_i)$;

(i) finally, $\mathbf{w}$ and $\mathbf{msg}$ depends on the **DCOP** solution method used to solve the problem. In this experiment, we assume **DSA** is used and set $\mathbf{msg}(c_i, c_j) = 1$ and $\mathbf{mem}(c_i) = |N(a_m)|$.

Notice that given our definition of the set of agents and their hosting costs, the initial distribution of computations always assigns exactly one computation to each agent.

Finally, we use **DRPM** on each problem / infrastructure combination, in order to achieve a 3-resilient system. This means that each agent will run one instance of **DRPM** for its computation in order to place three replicas.

Figure 5.12 represents the time required to place all replicas and achieve 3-resilience in the system. We can observe that replication is more difficult on our **IoT**-like infrastructure. This can be explained by the fact that, as communication costs are not uniform, **DRPM** generally needs to explore a larger part of the graph to find an low-cost place for the replicas. Scale free problems are marginally easier on average than random graph coloring problems, but also exhibit a larger variation between instances. In all cases, all instances are solved in very reasonable time, even for large problems, especially when considering that the replication of all computations only happens at startup and that the system does not need to wait for it for starting nominal operation.

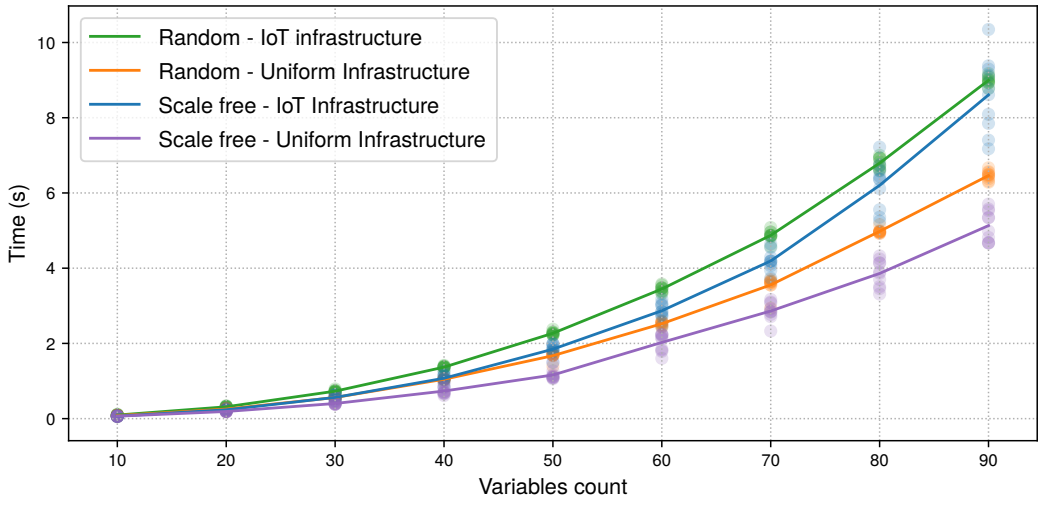


Figure 5.12 – Time for replicating computations for graph coloring problems

Figures 5.13 and 5.14 represent the number of messages and the total communication load induced by **DRPM**. The communication load is defined as the sum of the size of all messages exchanged during replication, and we count one symbol (cost, agent's name, etc.) for one unit. Scale free graph coloring problems on **IoT** infrastructure are clearly more communication-intensive, due to

the graph exploration as mentioned earlier, and also because messages grow larger when exploring the graph further from the agent, as they contain the longer paths.

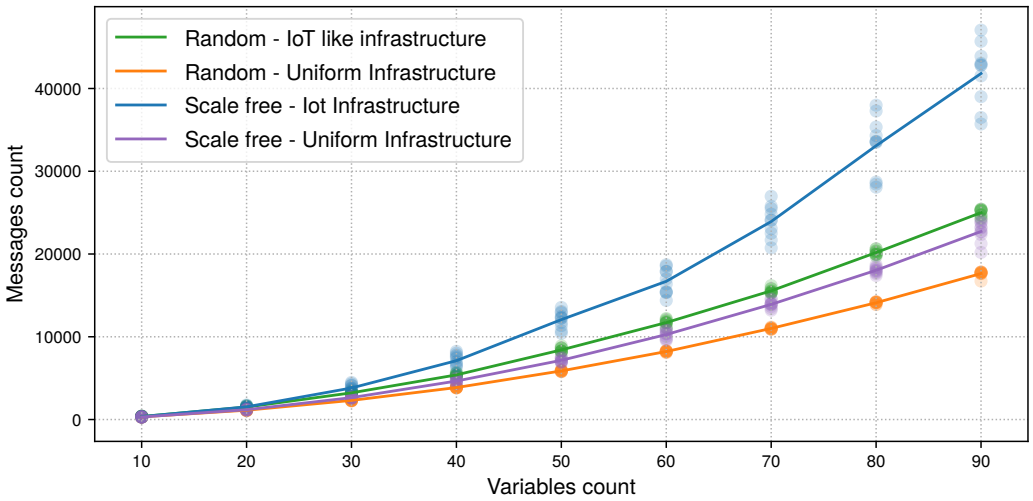


Figure 5.13 – Messages count when replicating computations for graph coloring problems

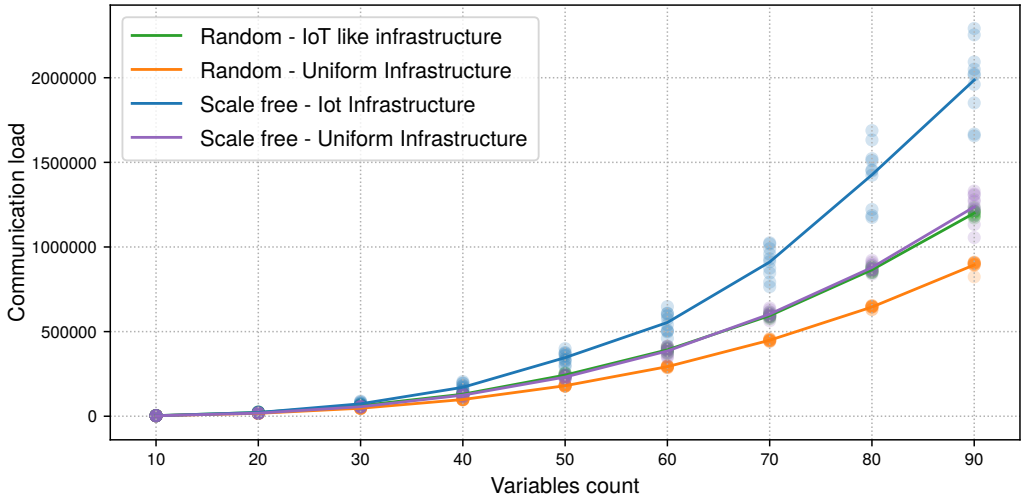


Figure 5.14 – Communication load when replicating computations for graph coloring problems

5.7.2.2 Evaluation of DRPM on SECP instances

We also evaluate **DRPM** on a set of **SECP** instances with increasing numbers of lights, physical models and rules, generated using the same protocol than described in Section 3.3.2: 10 instances are generated for each problem size.

We derive two computation graphs for each of these instances, for **DSA** and one for **MaxSum**, which are respectively a constraint graph and a factor graph based algorithm.

The multi-agent infrastructure used for these **SECP** problems is made of one agent for each light, and computations representing a light is assigned to the corresponding agent. Communication cost is uniform between agents and hosting costs are identical for all computation, except light

computation. The initial distribution is computed using **GH-CGDP**.

Figure 5.15 represents the time required to achieve 3-resilience on our **SECP** instances. As expected, replication is harder for **MaxSum**, as it requires more computations for the same problem. In any case, we argue that these replication times are perfectly reasonable for real-like system as this operation only needs to be run once when starting the system: during the nominal execution of the system, full replication is never needed and we can simply repair an existing replication if some agent fails or leave the system.

Figures 5.16 and 5.17 represent the total messages count and communication loads induced by **DRPM**.

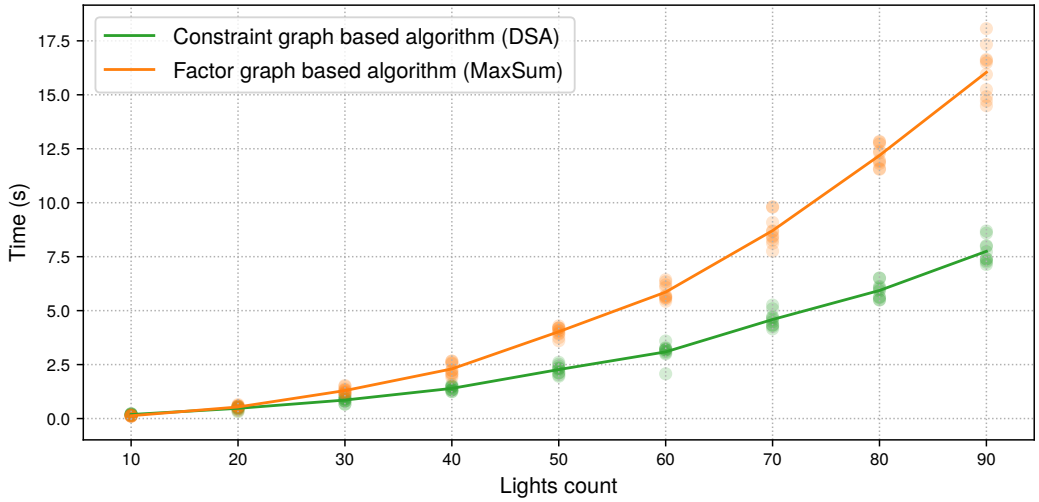


Figure 5.15 – Time for replicating computations for **SECP** instances

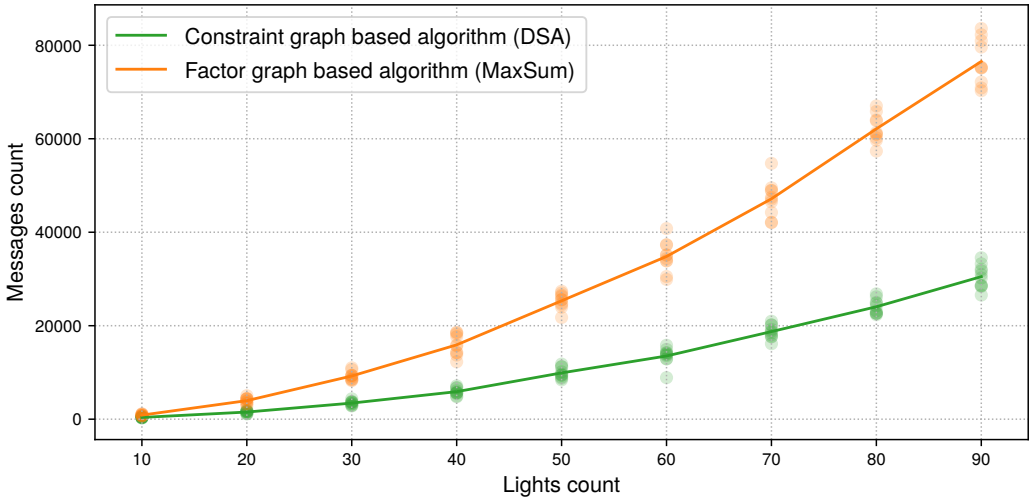


Figure 5.16 – Messages count when replicating computations for **SECP** instances

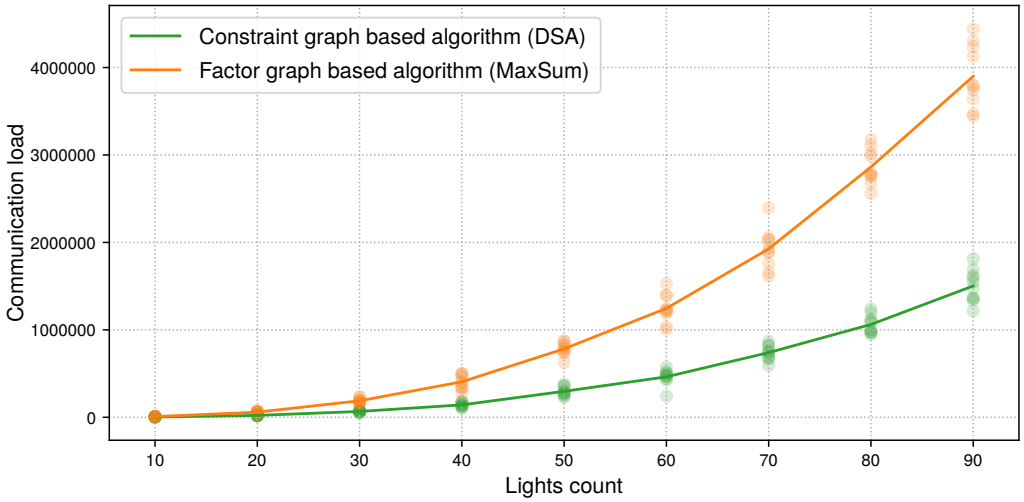


Figure 5.17 – Communication load when replicating computations for **SECP** instances

5.7.3 Evaluation of the DMCM Repair Method

In these experiments, we evaluate **DMCM**, our **DCOP**-based repair method. As **DMCM** models the repair process as a **DCOP**, we implement it using two different **DCOP** algorithms and compare their efficiency on this task.

We generate 20 random graph coloring problems and 20 scale free graph coloring problems, each with 25 variables. The problem generation is identical to the experiments on replication and is described in Section 5.7.2. A perturbation scenario is generated for each problem, made of 5 events where 2 random agents are removed from the system. We then derive for each problem two computation graphs, one for constraint graph and one for factor graph based **DCOP** algorithms.

As previously, we also generate two different agent infrastructures for each computation graph, an uniform infrastructure and an infrastructure designed to model the structure found in **IoT** systems (see Section 5.7.2). These infrastructures have one agent for each variable in the problem. The initial distribution of the computation graphs on these infrastructures is computed using the **GH-CGDP** heuristic (Section 4.5.4).

Finally, we distribute 2 replicas for each computation with **DRPM**.

During the experiments, perturbation events are injected in the system (i.e. agents are removed) and we run **DMCM** to repair the distribution. After the repair, the replica distribution is restored using **DRPM**, to ensure that no computation definition is lost. At the end of the scenario, 10 agents have been removed, from systems that were initially composed of 25 agents (40% of agents disappeared). Each scenario is executed 5 times and the results are averaged. Notice that during this experiment, the **DCOP** representing the graph coloring problem is not running, here we only analyse the repair of the computation graph representing it. Repair on a running system is investigated in Section 5.7.4.

We evaluate two different **DCOP** algorithms to implement **DMCM**: **DSA** and **MGM-2**. We use a synchronous implementation of these algorithms and allow them to run for 20 cycles. This allows us to compare the efficiency of our repair method, when using it on various problems,

representations and infrastructures, and to evaluate the time and communication load required for repairing systems.

		DSA	MGM-2
Uniform infrastructure	Constraint Graph	11%	8%
	Factor Graph	46%	13%
IoT infrastructure	Constraint Graph	36%	11%
	Factor Graph	35%	7%

Table 5.2 – Failure rates when using **DSA** and **MGM-2** for implementing **DMCM**

Table 5.2 shows the failure rates when using the two **DCOP** algorithms. Indeed, while repair is always successful for the first three perturbation events, it fails sometimes at the fourth or the fifth event. As a matter of fact, as we migrate the computations from removed agents to remaining agents, the average number of computation per agent rises, making the repair problem progressively harder as more computations must be migrated for a single perturbation event.

We can see that **MGM-2** is clearly better suited for implementing repair with **DMCM**, as it has a lower failure rate on all configurations. This can be explained by the fact that **MGM-2** is monotonic: as discussed in Section 5.5.5 it will handle hard constraints first, which ensures that all orphaned computations are hosted. When the repair fails, it is due to the limit on the number of cycles **MGM-2** is allowed to run. When raising that limit, the failure rates decreases. However, it is not possible to determine the number of cycles required for a repair operation, as in **MGM-2**, the 2-coordination mechanism is implemented by selecting a partner at random. We argue that it would be possible to design a variant of **MGM-2** better suited to our problem, by using agents characteristics when selecting a partner for coordination.

On the other hand, **DSA** generally has a high failure rate, as its stochastic behavior easily breaks these hard constraints. Raising the number of cycles as virtually no effect on its failure rate.

We cannot see any obvious relation between the failure rate and the infrastructure (IoT or uniform) or the computation graph representation (constraint graph or factor graph).

Figures 5.18 and 5.19 show the time and communication load of repair operations when using **DSA** on random and scale free graph coloring problems. As already explained, we can clearly see that repair is more difficult after several events, it takes more time and induces more communication load, as more computations must be migrated on average. We can also see that repairing scale free problems takes less time than for random graph coloring problems, and induces a slightly lower communication load.

Problems represented as constraint graphs are also clearly easier to repair that when using factor graphs, which makes sense as the factor graph representation requires more computations.

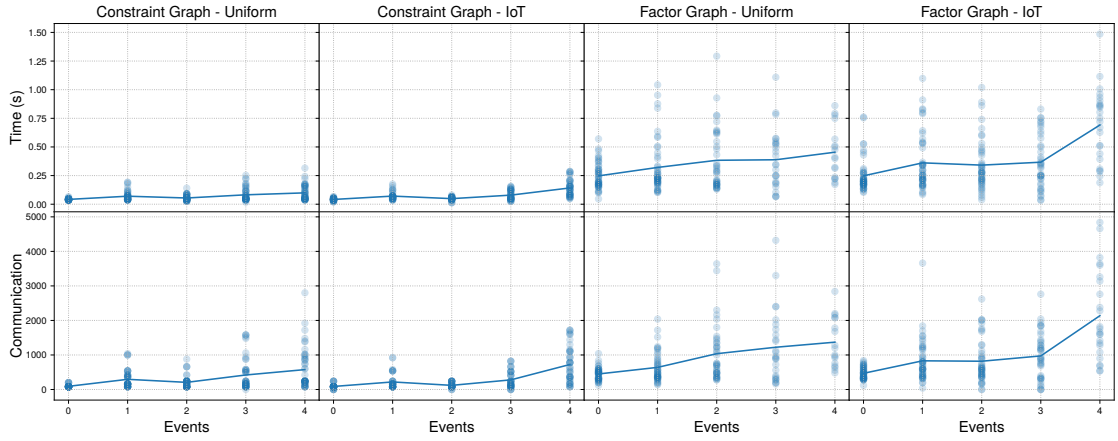


Figure 5.18 – DMCM repair using **DSA** on random free graph coloring problem

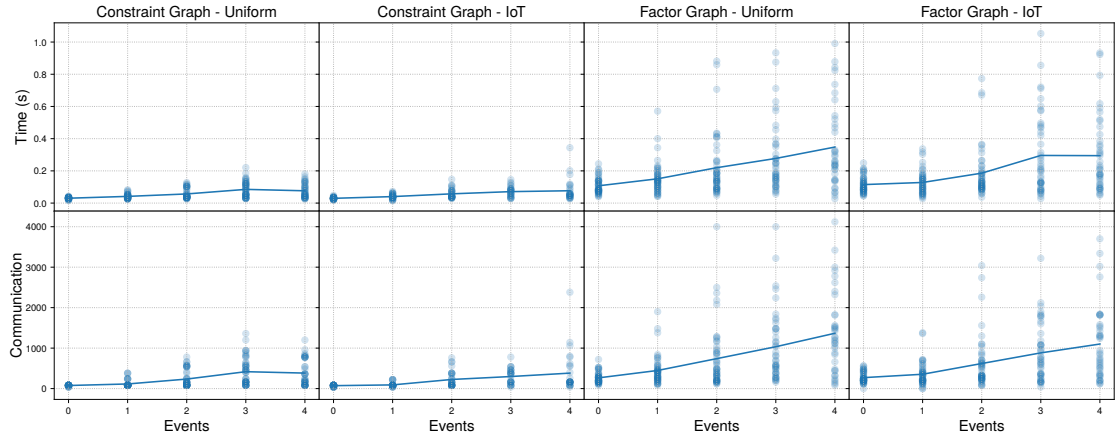


Figure 5.19 – DMCM repair using **DSA** on scale free graph coloring problem

Figures 5.20 and 5.21 show the time and communication load of repair operations when using **MGM-2** on random and scale free graph coloring problems. The overall behavior is similar to what we observed with **DSA**.

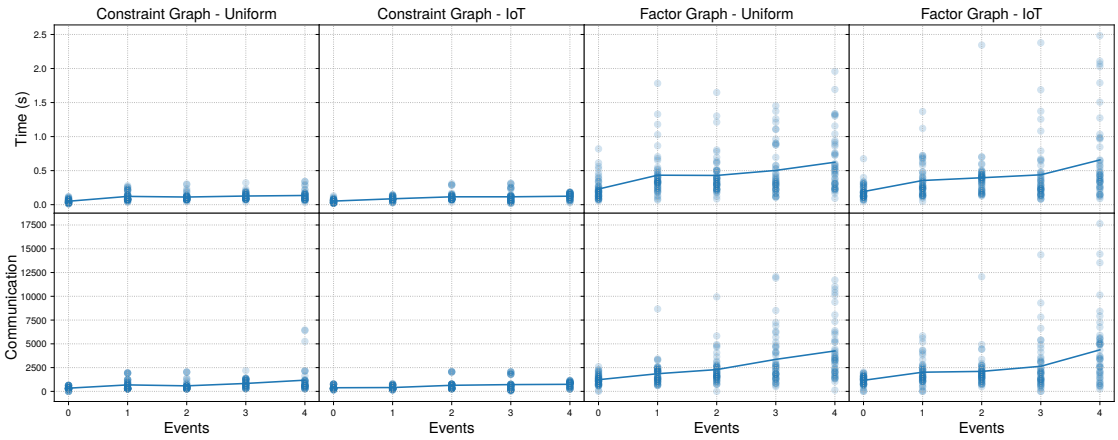


Figure 5.20 – DMCM repair using **MGM-2** on random graph coloring problem

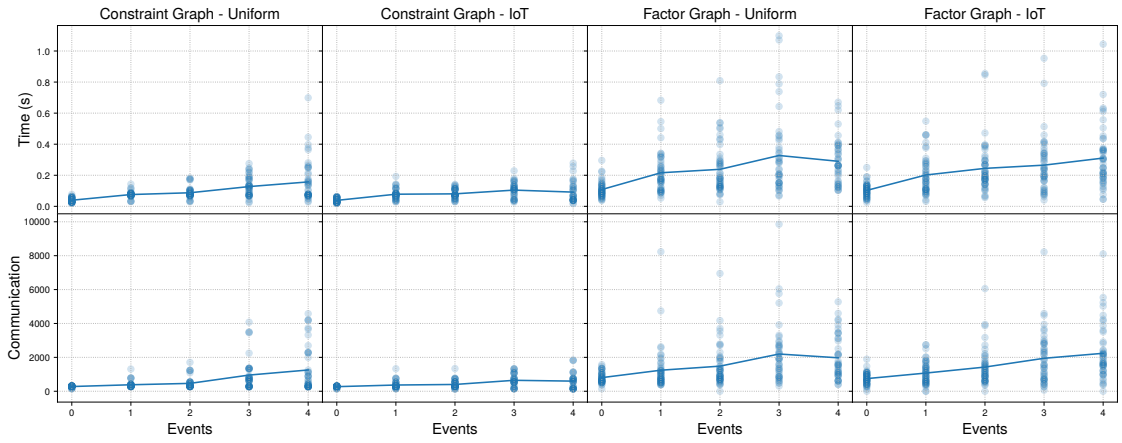


Figure 5.21 – DMCM repair using **MGM-2** on scale free graph coloring problem

When comparing **DSA** and **MGM-2**, we can see that **MGM-2** requires more time and communication load to complete a repair operation. However, **MGM** extra cost is still very acceptable, especially when looking at the lower failure rates it achieves, as discussed previously. Therefore, we argue that **MGM-2** is better suited for implementing **DMCM**.

5.7.4 Resilience

In this section we evaluate the efficiency of **DRPM**[**DMCM**], our solution method for k-resilience, on *running* systems. Given the results of our experimental evaluation of **DMCM**, we elect to use **MGM-2** as our repair **DCOP** algorithm. Thus, all experiments in this section are performed using **DRPM**[**MGM-2**].

5.7.4.1 Evaluating Resilience on Benchmark Problems

First, we evaluate our approach for $k$resilience, **DRPM**[**MGM-2**], on benchmark graph coloring problems. As previously, these problems are composed each of three components: a graph coloring problem definition (scale free or random), a multi-agent infrastructure (uniform or IoT), and a disturbance scenario. Problems generation is performed using the same method than we used in previous experiments and is described in Section 5.7.2.

The graph coloring problems are solved with **A-DSA** and **A-MaxSum**, and agents are removed at runtime, during the resolution process. We selected these **DCOP** algorithms because they are both asynchronous and support message loss. Furthemore they can be considered as *staleless* (for **A-DSA**) or *almost stateless* (for **A-MaxSum**), which is necessary for applying **DRPM**[**DMCM**], as stated in Section 5.5.1.

After each removal, the system is repaired using **DRPM**[**MGM-2**]. Practically speaking, we use a **DCOP** (solved with **MGM-2**) to repair and restore nominal operation on another **DCOP** (solved with **A-DSA** or **A-MaxSum**), which represents the initial dynamic problem we want to solve (in this case, a graph coloring problem). Additionally, after each repair we re-run **DRPM**, for migrated computations and computation whose replica were hosted on removed agents. This ensures that each computation in the system still has k replica.

Additionally, we also solve the same problems without any disturbance, in order to assess the impact of our repair method on the quality of the solution returned by **A-DSA** and **A-MaxSum**.

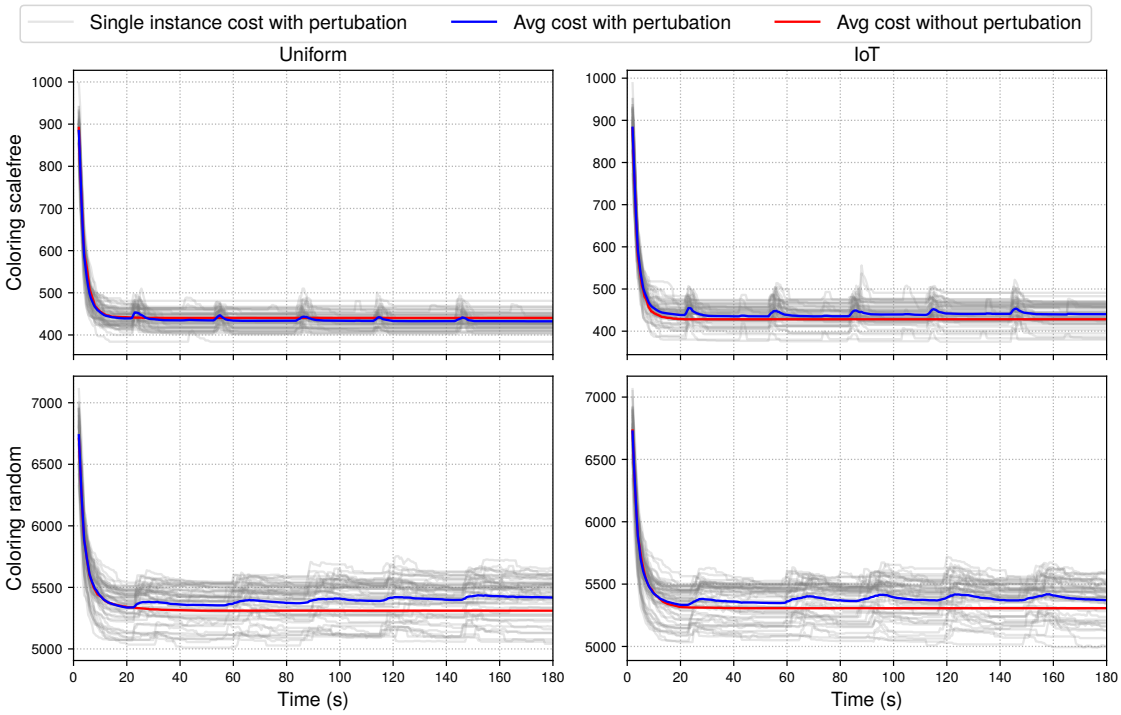


Figure 5.22 – Cost of **A-DSA** solution at runtime, with (blue) and without perturbation (red), on uniform (left) and IoT-like (right) infrastructure, and when solving scale free graph coloring (top) and random graph coloring (bottom), using **DRPM**[**MGM-2**] to repair

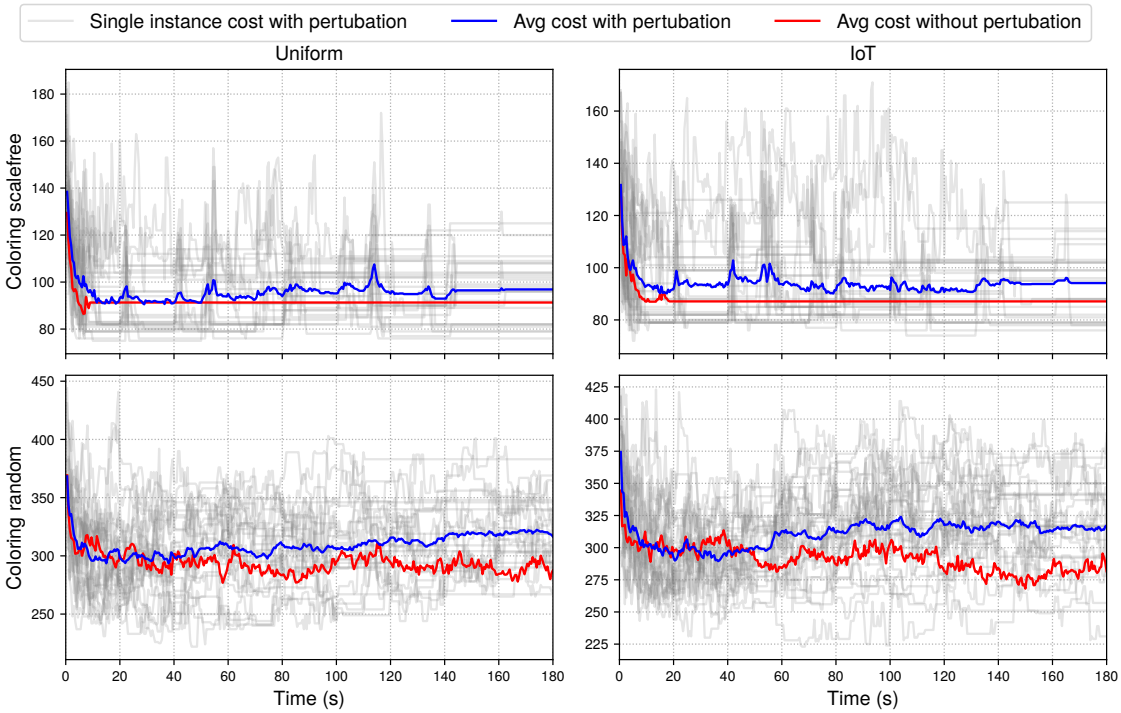


Figure 5.23 – Cost of **A-MaxSum** solution at runtime, with (blue) and without perturbation (red), on uniform (left) and problem-dependent (right) infrastructure, and when solving scale free graph coloring (top) and random graph coloring (bottom), using **DRPM**[**MGM-2**] to repair

When using **A-DSA** to solve the graph coloring problems, we generate problems with 100 variables and place 3 replicas for each computation, in order to achieve 3-resilience, Disturbance scenarios are made of 5 events, where 3 agents are removed. Figure 5.22 shows the cost of the solution found by **A-DSA** over time. The cost of each of the 100 runs is displayed in transparent grey, the overall shapes illustrates the fact that the system's behavior is consistent across the various instances. The average solutions costs with (in blue) and without (in red) perturbation are also plotted.

We can see that the solutions on the disturbed system degrade when agents are removed, but quickly improve again when the system recovers. Here, the replicas that are activated by the repair process, as opposed to the computation that were hosted on removed agents, do not need accumulated knowledge to recover a consistent state, thanks to message passing with neighbors. In **A-DSA**, computations gather new information about costs from their neighbors at each message exchange; as a consequence migrated compuation are able to very quickly select an appropiate value, based on the value of their neighbors.

Interestingly, we can see that the pertubations even allow **A-DSA** to reach a better average solution quality for scale free problems on a uniform infrastructure. Indeed, **A-DSA** is a local seach algorithm that can get trapped in local optimal and the perturbation can help mitigating this issue by resetting several values simultaneously.

When using **A-MaxSum** to solve our graph coloring problems, we generate problems with 25 variables and place 2 replicas for each computation, in order to achieve 2-resilience. Disturbance scenarios are made of 5 events where 2 agents are removed. Notice that we consider here smaller problems than for **A-DSA**, since **A-MaxSum** operates on factor graph, which requires more computations (one more per edge in the graph) than the constraint graphs used by **A-DSA**. Still, on random graph with density of 0.3, such problems require on average $25 + 0.3\frac{25\times24}{2} = 125$ computations to manage, which is roughly similar to the number of computations induced by **A-DSA** when solving problems with 100 variables.

In Figure 5.23, we can see that the solutions on the disturbed system degrade when agents are removed, but improve again when the system recovers, as for **A-DSA**. However, **A-MaxSum** operation on very cyclic problems like random coloring is known to be very noisy, even using a high damping factor (here we use 0.8), as proposed in [23]. Moreover, belief propagation algorithms like **A-MaxSum**, computations are not really stateless: they accumulate information about constraints and preferences from their neighbors. When activating a replica, the new active computation starts afresh and an indeterminate number of message rounds are needed to restore that information. All in all, **A-MaxSum** operation is more impacted by the perturbations and repair procedure than **A-DSA**.

In order to evaluate the quality of the repaired distributions of computations, we also measure the degradation of the distribution all along the system lifetime. At each event, we assess the cost of the current distribution of the computation graphs (for A-DSA) against the initial distribution cost (which is optimal, but cannot be computed at runtime). Figures 5.24 and 5.25 show the distribution costs for the 100 runs. As the global distribution cost is made of communication and hosting costs, we also plot these two costs independently.

In every case, the hosting cost logically increases by $10 \cdot k$ at each perturbation event, as k

computation are moved from their initial agent to another (where the hosting cost is 10). On scale free models, hosting costs and communication costs have the same order of magnitude. But, for random graphs, higher density implies that there are more edges in the graph and as a consequence the overall communication cost is higher. In general, the communication costs decrease at each repair, as computations can move to a less expensive agent.

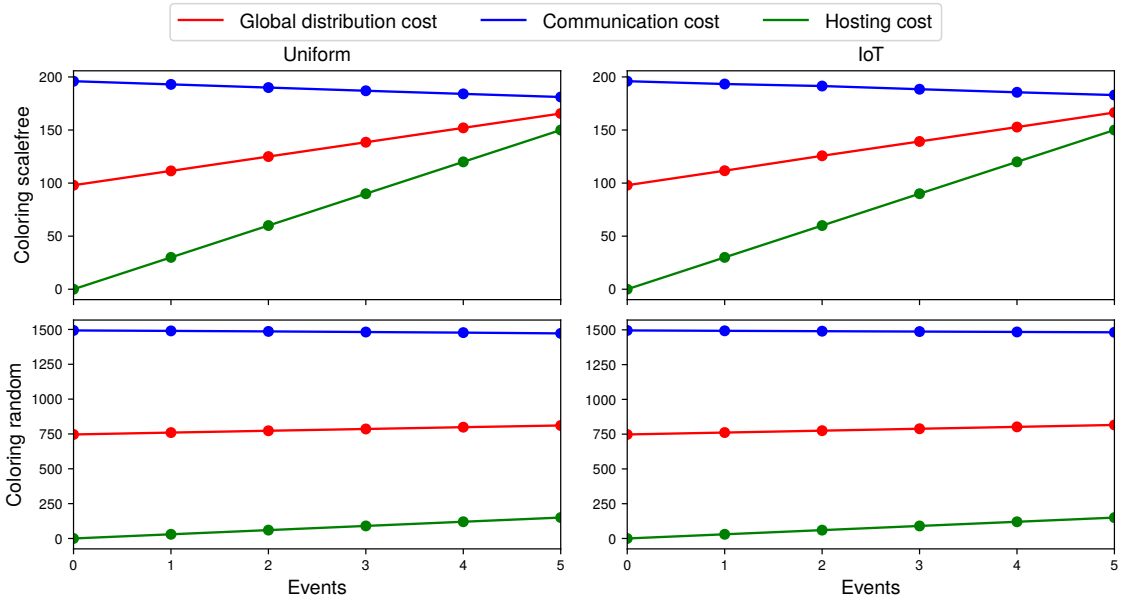


Figure 5.24 – Cost of the distribution of computation graphs on which **A-DSA** operates, after each event, on uniform (left) and problem-dependent (right) infrastructure, and when solving scale free graph coloring (top) and random graph coloring (bottom) problems, using **DRPM[MGM-2]** to repair

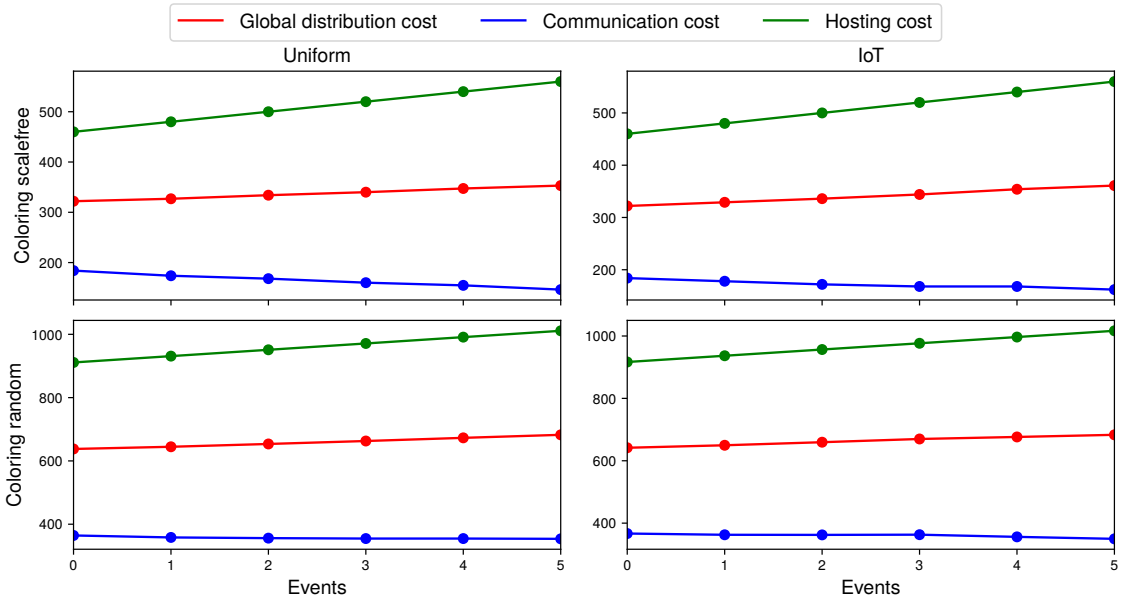


Figure 5.25 – Cost of the distribution of computation graphs on which **A-MaxSum** operates, after each event, on uniform (left) and problem-dependent (right) infrastructure, and when solving scale free graph coloring (top) and random graph coloring (bottom) problems, using **DRPM[MGM-2]** to repair

5.7.4.2 Evaluating Resilience on SECP

We now evaluate the efficiency of **DRPM[MGM-2]** on running **SECP**. The problem generation method is identical to what we used when evaluating static **SECP** (Section 3.3): we generate 100 **SECP** instances with 30 lights, 9 physical models and 6 rules. One scenario of 5 events is generated for each instance, at each event 2 random agents (i.e. light device) is removed.

The intial distribution of the computation graphs derived from these problems is computed with **GH-CGDP**.

The 100 **SECP** instances are solved with **A-MaxSum** and **A-DSA**, as in our previous experiments on graph coloring problems, since these algorithms support message loss. However, the **A-MaxSum** implementation we use here is customized to improve its behavior after a repair operation: once orphaned computations have been migrated and restarted on an active agent, the accumulated cost table of their neighbors is flushed. Additionally, the standard mechanism used to avoid sending duplicate messages (classicaly used to detect termination when messages converge [36]) is inhibited and belief propagation is restarted. Notice that this can be implemented in a distributed manner with a simple token passing approach.

During the solving process, we inject the scenario's events in the system every 30 seconds, each time removing 2 agents, and repair the system using **DRPM[MGM-2]**. After each repair we re-run **DRPM**, for migrated computations and computations whose replica were hosted on removed agents. This ensures that each computation in the system still has k replica.

As previously, our mechanism amounts to using a **DCOP** (solved with **MGM-2**) to repair and restore nominal operation on another **DCOP** (solved with **A-DSA** or **A-MaxSum**), which represents the initial dynamic problem we want to solve (in this case, **SECP**).

Additionally, we also solve the same problems without any disturbance, in order to assess the impact of our repair method on the quality of the solution returned by **A-DSA** and **A-MaxSum**.

Figures 5.26 and 5.27 show the cost of the solutions found by **A-DSA** and **A-MaxSum** over time. As the **SECP** model contains both soft and hard constraints, we plot separately the number of violated hard constraints (bottom) and the sum of costs of the soft constraints (top) The results for each of the 100 instances are displayed in transparent grey and the average cost across all instances is plotted in blue. The average cost of the same instances solved without disturbance is plotted in red, but is barely visible as the average repaired cost is extremely similar.

We can see that both **A-DSA** and **A-MaxSum** behave remarkably after a repair: during a short period after the repair the solution cost and the number of violated hard constraints increase, but quickly get back to the quality level achieved before the agents were removed. Overall, we can say that **DRPM[MGM-2]** is well suited for repairing running **SECP** and that the quality of the solution produced over time is barely affected by the repair operations.

However, when comparing results produced by **A-DSA** and **A-MaxSum**, we can observe that **A-DSA** yields higher costs, which coincides with what we obtained when experimenting on static **SECP** in Section 3.3.

After a repair, **A-MaxSum** generally breaks more hard constraints than **A-DSA** but we argue that it is not really problematic as it always manage to get back to the level it had before the

disturbance. Notice however that the average number of hard constraints violations is not equal to zero. Indeed, there are some instances where **A-MaxSum** struggles with hard constraints, as we already experience with static **SECP** (see Section 3.3).

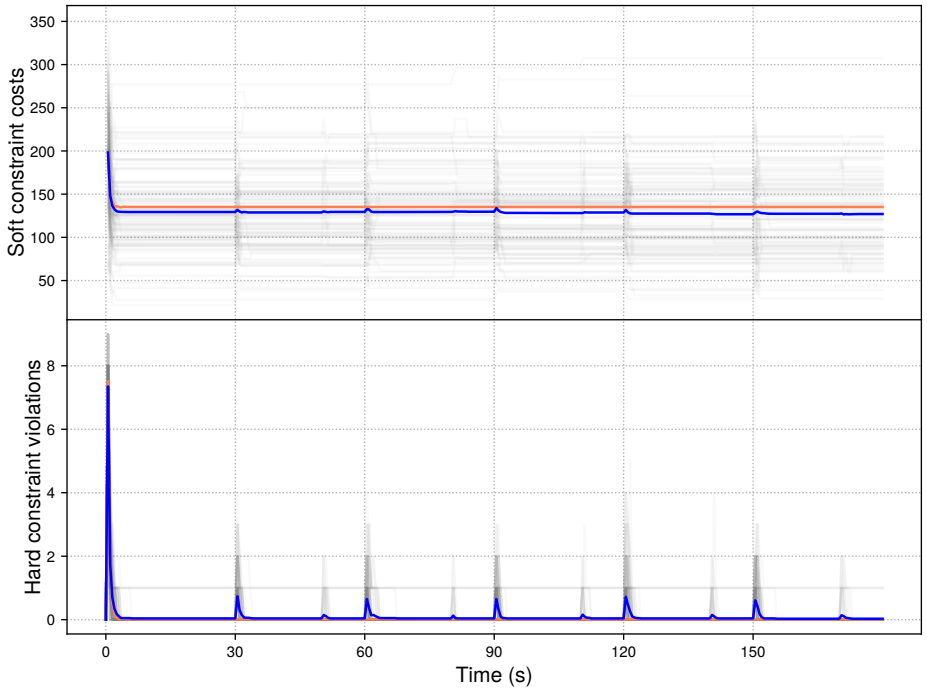


Figure 5.26 – Cost and hard constraints violations of operating **A-DSA** to solve **SECP**, repaired with **DRPM[MGM-2]** (blue: with perturbations, red: without perturbation)

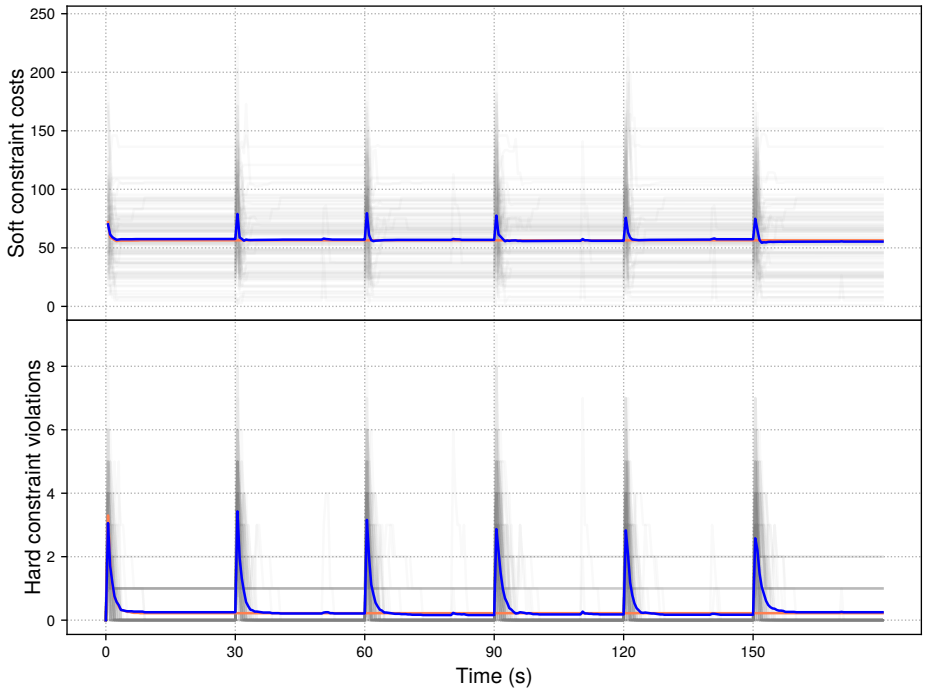


Figure 5.27 – Cost and hard constraints violations of operating **A-MaxSum** to solve **SECP**, repaired with **DRPM[MGM-2]** (blue: with perturbations, red: without perturbation)

5.8 Summary

In this chapter, we demonstrated that **SECP**, like most problems in **AmI** and **IoT** (and probably in other domains as well), is in fact a dynamic problem. After expounding a synthetic state-of-the-art on **Dyn-DCOP**, we explained that dynamics in these systems could be classified in two categories: computation dynamics and infrastructure dynamics.

We argued that computation dynamics could be handled at the **DCOP**-algorithm level and a reactive approach was appropriate for our use cases. On the other hand, infrastructure dynamics generally require to revise the distribution of the computation graph. Thus, we introduced several approaches to revise the distribution in the case of agent(s) arrival and departure. Most notably, we introduced the idea of k-resilience, which characterises a system able to survive the failure of up to k agents, and proposed **DRPM**[**DMCM**], a technique to implement it.

We also evaluated experimentally our repair methods, both for single agent arrival and departure and k-resilience. Based on this results, we are confident that **DRPM**[**MGM-2**] can be realistically applied on **SECP** and that **A-MaxSum** and its derivatives are viable candidates for solving these problems in dynamic settings.

DRPM[**DMCM**], and more generally k-resilience, have mostly been defined for computation graphs used to solve **AmI** and **IoT** problems modeled as **DCOP**s. However, these mechanisms could be applied, and potentially adapted, to many other distributed approaches where computation graphs are used, like Dataflow Models or BSP, as briefly presented in Section 4.5.

6 Studying DCOP for IoT Systems Using pyDCOP

In order to design and evaluate experimentally the solution methods presented in previous chapters, we required a software library. After considering the various options available in the **MAS** ecosystem, we decided to develop our own software library for that purpose. In this chapter, we explain the reasons for this choice, including a brief presentation of the major existing frameworks, and introduce pyDCOP, our open source library designed to foster the study and research on **DCOP**.

6.1 Implementing Multi-agent Systems

Many software libraries, most of which originate from academic research, are available in the **MAS** ecosystem. Given the numerous different approaches and application domains of **MAS**, these solutions cover very various needs and target different uses cases and communities. Some solutions propose full-stack integrated environments, and can be used to implement all the components in a **MAS** while other target one specific aspect (agent behaviour, communication, organization, etc.) and must be used in collaboration with other solutions when developing a full system.

6.1.1 Frameworks from other MAS Perspectives

When investigating the solutions developed in other **MAS** communities, we quickly realized that they were not ideal for implementing our approaches. These solutions can be categorized, not exhaustively, into the following families:

Simulation Frameworks. A family of solutions focus on the use of **MAS** for simulation purposes. This family is very active and mature, with several commercial offerings, and includes tools like Gama [6], Cormas [17], NetLogo [147], IODA[67] and many others. However, they are also not suited to our needs, as we also intend to implement real physical distributed systems.

Agent Programming Libraries. Several **MAS** communities advocate that programming languages designed specifically for agents are required; this approach is usually denoted as *Agent Oriented Programming*. Libraries from this family include Jason [15], SARL [117] and Jadex [25].

We argue that Distributed Constraints Reasoning algorithms are better developed with traditional programing language and that such agent-specific languages are not required in our case.

Interaction Oriented Libraries. Some other libraries like SACI [54], MadKit [47], IODA [67], CArtAgO [114] or Jade [9] focus on the interactions among agents (and their environment) in a **MAS**, which includes communications but also organisational aspects. These libraries generally do not impose any internal agent's structure, leaving their users decide on the model they want to use. While these libraries could be used to implement communication in our systems, that would still leave much work to be done and we feel that the gain would be to small to justify the efforts required to integrate them with our solution methods.

Multi-agent Programing Frameworks Finally, some solutions like JACK [4] and JaCaMo [14] provide full environments for the engineering of a **MAS**, including all related concepts like organizations, environment representation, etc. These frameworks are often based on several more specialized libraries; it is for instance the case of JaCaMo, which combines Jason, CArtAgO and Moise [53]. However the high level of integration of such platforms makes it difficult to reuse them for implementing approaches they were not designed for. For example, **BDI** model [113] based frameworks like JACK and JaCaMo cannot be easily adapted to the agent's model of the **DCOP** framework, which is algorithmic and not plan-based like **BDI**. Besides, they include many concepts that we would not use and cannot produce the metrics needed to evaluate our solution methods.

6.1.2 DCOP Libraries

Faced with the available libraries we mentioned in the previous section, the Distributed Constraint Reasoning community has developed over the years several libraries specifically tailored to its needs. We now list the existing solutions in this area, along with their respective strengths and limits (in our opinion and to the best of our knowledge).

AgentZero is a Java-based library developed at the Ben-Gurion University which supports a large set of functionalities for the study of Distributed Constraints Reasoning algorithms [74]. Unfortunately documentation is scarce and the source code repository [1] has not been updated since 2016. Besides, this repository does not contain the implementation of any **DCOP** algorithm, but only the code of the infrastructure provided by AgentZero.

Frodo2 [70] is actively developed [2] by the Artificial Intelligence Laboratory (LIA) of École Polytechnique Fédérale de Lausanne (EPFL) and is probably the most commonly used library for evaluating DCOP algorithms. While being very well engineered and providing numerous DCOP algorithmic implementations, it does not provide the required features to study and prototype DCOP in a dynamic system like IoT.

1. https://github.com/bennylut/agent-zero
2. https://frodo-ai.tech/

DisChoco [145] is also Java-based and exhibits an interesting modular design. It supports real distributed settings through the use of SACI [54] for the communication layer. However, the project [3] seems to be discontinued and has not been updated since 2014.

DCOPolis [137] is a very rich java-based library that contains many implementations of standard **DCOP** algorithms. It supports for the concepts of *virtual agents*, which can be seen as a first step toward the idea of distribution we explored in our work. Unfortunately, the project has not been updated since 2009 and seems to be discontinued. The currently available source code is nonetheless, in our opinion, one of the best references available for many algorithms.

A few repositories of **DCOP** implementations are also available, like *USC Distributed Constraint Optimization Problem (DCOP) Repository* [4] or JSAM [5] However, while still useful, these repositories simply list implementations, potentially developed in different programming languages, and do not provide integrated frameworks that could be used to produce metrics and compare algorithms.

6.2 pyDCOP at a Glance

Following the review of the existing libraries and frameworks for **MAS**, we draw the following conclusions:

- Tools from other **MAS** communities are not well suited for **DCP**, dedicated tools are required. All of existing DCOP libraries only provide implementations for a small subset of the algorithms proposed other the years by the community.
- Most research in this domain is done on closed code bases, without publishing any implementation of algorithms, which hinders research.
- No existing libraries can be used to study some of the concepts we are focusing on, namely distribution, dynamics and resilience. These concepts are of paramount importance when applying **DCOP** in **IoT** settings.
- No library focused on **DCP** and **DCOP** currently exist in python, even though this programing language benefits from a huge ecosystem of libraries for science and research.

Based on these conclusions we decided to develop our own library for Distributed Constrained Reasoning, pyDCOP, in order to design, implement and evaluate our solution methods.

pyDCOP has been open-sourced in 2017 and is in continuous development. It is available at `https://github.com/Orange-OpenSource/pyDcop` and provided with a comprehensive documentation [6] including a reference manual and tutorials, which have already been used for lectures and conference tutorials.

pyDCOP provides implementation for many classical **DCOP** algorithms, including **DSA** [158], **A-DSA** [40], **DBA** [155], **GDBA** [91], **MGM**, **MGM-2** [77], **DPOP** [104], **ADOPT** [86],

3. `http://dischoco.sourceforge.net/`
4. `http://teamcore.usc.edu/dcop/`
5. `https://github.com/coenvl/jSAM`
6. `https://pydcop.readthedocs.io/en/latest/`

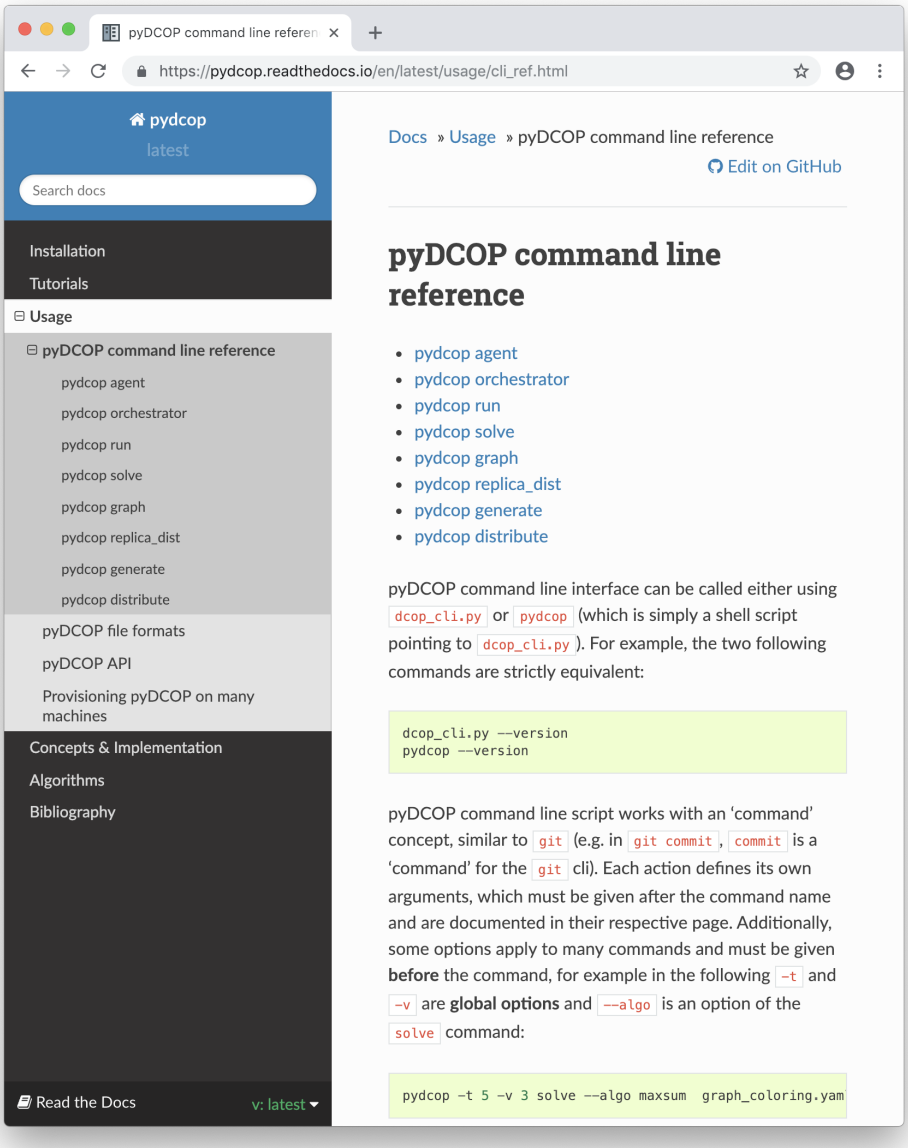


Figure 6.1 – pyDCOP extensive documentation

SyncBB [49], **NCBB** [21], **A-MaxSum** and **MaxSum** [36], but it also allows the rapid development of new algorithms. Notice that while pyDCOP focuses on optimization problems, it can also be used for Distributed Constraint Satisfaction.

When studying an existing algorithm or developing a new one, pyDCOP provides all the needed infrastructure: thanks to the numerous base classes and 'plumbing' utilities one can simply focus on its algorithm design. The modular architecture of pyDCOP –which decouples communication, agent managements, and algorithmic utilities– ensures that algorithms will be able to run in the several runtime environments and settings supported by pyDCOP.

When running an algorithm, various metrics can be produced and used to benchmark algorithms or the effect of meta-parameters in a specific problem topology. These metrics notably include runtime, number of cycles, number and size of messages, and cost and quality of the solution.

In addition to state-of-the-art **DCOP** algorithms, pyDCOP also includes the approaches presented in this thesis to apply the **DCOP** framework to dynamic systems like the IoT. The distribution of **DCOP** computations (see Section 4.1) is San issue that received little attention so far but is paramount when working on real-world problems. pyDCOP provides implementations for all distribution methods presented in Chapter 4: **GH-SECP-CGDP**, **GH-SECP-FGDP**, **ILP-SECP-CGDP**, **ILP-SECP-FGDP** , **CGDP** and **ILP-CGDP**. In IoT systems, the devices are typically very constrained (both CPU and memory wise), and the network is generally also considered to be a costly and limited resource. As a consequence, pyDCOP's distribution mechanisms take these elements into account and produce distributions that optimize for network communication while ensuring the agents capacities are respected.

Resilience is also a key issue when building a **MAS**. In dynamic environments the problem may evolve at runtime and agents could join and leave the system unexpectedly at any time. In order to ensure resiliency, pyDCOP implements **DRPM**[**DMCM**], the self-repair mechanism introduced in Section 5.5, and is able migrate the computations needed to solve the **DCOP** from on agent to another. For that purpose, pyDCOP implements **DRPM**, a distributed replication mechanism inspired by distributed databases and presented in Section 5.5.3, which makes sure that the definition of the problem is not lost when some agents leave the system. Based on these two mechanisms, in case of an agent failure, remaining agents can self-repair the system by migrating orphaned computations to the remaining agents. This self-repair function is also modeled as a **DCOP**, where agents cooperatively agree on the best place to host the repaired computations required to solve the initial problem.

pyDCOP can be used through a powerful command line interface, which allows running systems with many agents and comes handy when scripting complex benchmarks. pyDCOP also provides a graphical interface, for demonstration and prototyping purposes, which can display in real-time the current state of an agent or the whole system. This user interface is implemented as a web application, which can be displayed either in the device running the pyDCOP agent, or on any other computer in case the device does not have a screen.

The agents solving the problem can run on the same machine and even in the same process, using in-memory communication, which is convenient during development but also allows large-scale systems. They can also run on different computers, communicating over the network, for prototyping real distributed systems. pyDCOP is multi-platform and can run on Windows, Mac and Linux. Scripts are also provided to ease the deployment of agents on many computer, typically virtual machines or single-board computers like Raspberry Pis.

6.3 pyDCOP Concepts and Architecture

As presented in Section 2.3.2, Definition 2, a **DCOP** is traditionally represented with a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$ where $\mathcal{A}$ represents a set of agents. pyDCOP's architecture is based on this formal representation and extends it using the concept of *computations* introduced in Section 4.5.1. pyDCOP manages a set of *software agent* objects, which coordinate cooperatively, using message-passing algorithms. This coordination is implemented in computations, which are the algorithm's building blocks, *hosted* on agents.

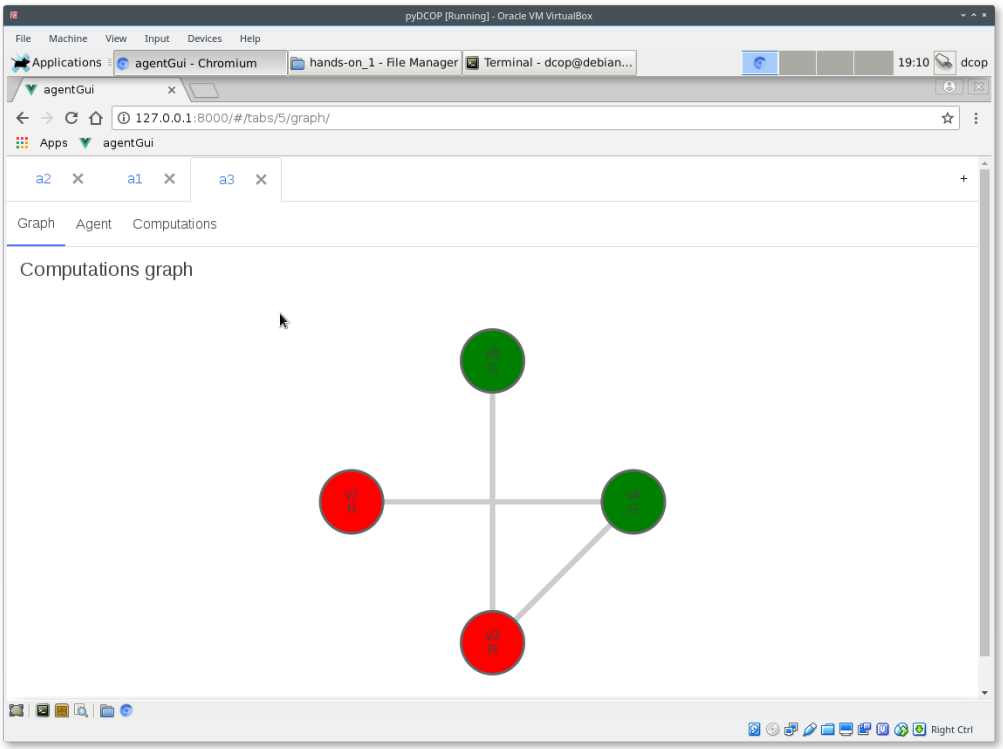


Figure 6.2 – pyDCOP Web UI to access agents' inner state

Figure 6.3 depicts this architecture: each agent run independently, using a runtime environment (see Section 6.3.3) and can only communicate with agents whose address it knowns.

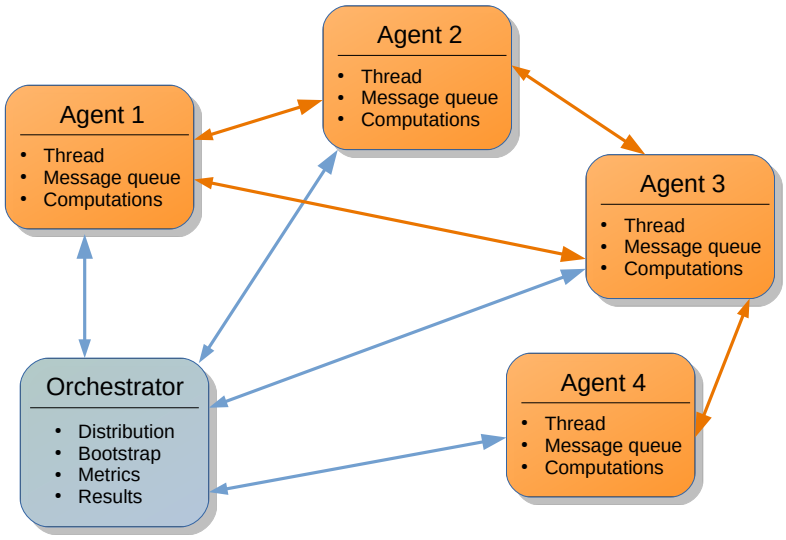


Figure 6.3 – Sample pyDCOP Architecture Instantiation

One specific agent, called the *Orchestrator*, is responsible for administrative tasks in the system like collecting metrics or bootstrapping the system. For instance, it informs at startup each agent of the variables it is responsible for and the constraints they are involved in.

Additionally, the Orchestrator agent provides a directory service, which agents can use to find the address of other agents. This is particularly useful when dealing with dynamic systems where new agents, whose address cannot be known at startup, can join at any time. Such discovery mechanism could also be implemented in a distributed manner but we decided to focus on distributed decisions making and avoid the unnecessary (for our studies) implementation complexities it would have induced. Reliable, robust and distributed discovery and directory services would however be a very interesting and useful area for future research.

Notice that this Orchestrator agent is the only centralized element in the system but never participate in any collective decision-making and is only an implementation artifact; it could actually be removed at runtime without impacting the nominal execution of the system, but the system would then not be monitored, which is generally required when running experimental evaluations.

6.3.1 Communication

All communication between agents is based on a message-passing mechanism. Each agent has its own message queue and handles messages sequentially, ensuring that agents only process one message at a time. Messages in the queue are prioritized, which allows handling urgent operations first. For example, messages used in the sel-repair mechanism use an higher priority to restore nominal operations as fast as possible, while metrics are collected with low-priority messages to avoid interfering with de decision making process.

Figure 6.4 represents this inter-agent communication scheme.

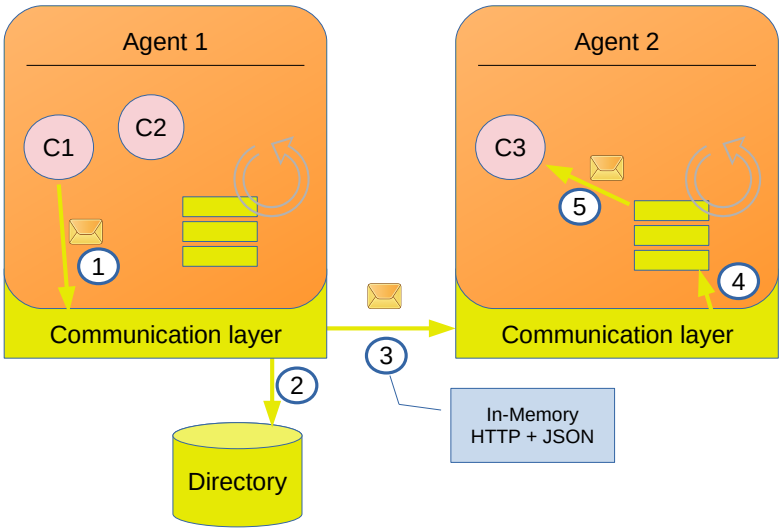


Figure 6.4 – pyDCOP Inter-agent Communication Scheme

Depending on the runtime used (see Section 6.3.3), communication is implemented with in-memory message passing or HTTP requests (with payload encoded in JSON). In-memory messages allow simulating systems with many agents, with a reasonable overhead, on a single computer while http messaging supports distributing agents on several computers or connected devices. Other communication mechanism could easily be implemented as the communication layer is completely

decoupled from the agents, who totally ignore the concrete implementation currently used. One could for example develop a communication layer targeting low-power networks such as **6LowPan**, using **CoAP** and BSON [7] encoding.

6.3.2 Inner-agent Architecture

Figure 6.5 depicts the inner-architecture of agents. All the agent's behaviors are implemented through *computation*. Of course, this includes computations that implement **DCOP** algorithms, but also administrative tasks and utilities like replication (for resilence implementaion), metrics collection, graphical interface serving, remote management, etc.

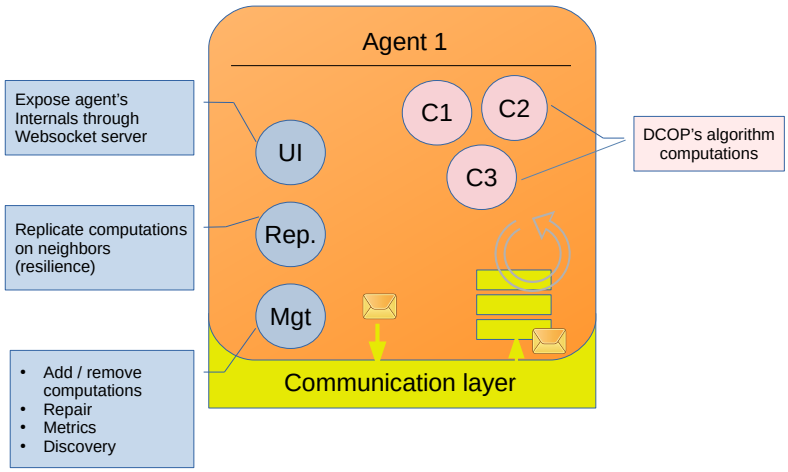


Figure 6.5 – pyDCOP Agent Architecture

As agents can only interact through messages and each agent run independently, processing messages one at a time, we avoid synchronization issues and are guaranteed that no race condition can happen.

When an agent receives a message, it simply dispatches it to the target computation, which can update its internal state, send new messages to other computation when necessary and even create new computation on this agent (that is for example how self-repair and computation migration is implemented).

Notice that this approach is very similar to the actor model [48] for concurrent computation from distributed computing. Indeed, we believe that an actor model implementation could be a very good low level implementation for a **MAS** library, especially when studying **DCOP**. The only reason we did not base pyDCOP on such implementation is that no suitable library for the actor model is available in python.

7. http://bsonspec.org/

6.3.3 Runtime Environments

When working on a problem, pyDCOP runs as many agents than specified by the problem. Each of this agents execution is controled by a *runtime environment*.

pyDCOP provides three runtime environments:

- The **process environment** uses one separate process for each agent. Agents can live on different computers and communicate using the HTTP communication layer. This mode is ideal for prototyping real systems, for example using low-power computer like the Raspberry Pi, but is more complex to implement. Besides, the communication layer induces an overhead which, while realistic, can prove problematic when running large scale simulations.
- The **thread environment**, where each agent is associated to one thread, which is used to run message handling procedures defined by the computations. When using this environment, pyDCOP runs as a single process (and thus on single computer), and uses in-memory message passing. This allows running large systems (several dozen of agents) with reasonable overhead and is the mode commonly used when running simulations. However, due to notoriously bad handling of thread concurrency by the python interpreter [8], performances can be severely reduced when scaling to larger systems.
- The **executor environment** (still in development) mitigates the performance issues of the thread environment by sharing a thread between several agents. It is inspired by the implementation of actor model found in the akka [9] framework.

6.4 Using pyDCOP

In this section, we present several usage patterns for pyDCOP, both from a simple user and a developer point view, and illustrate these usages with several examples.

6.4.1 File Formats

pyDCOP uses its own YAML-based [10] file format for defining **DCOP**. This format supports defining domains, variables, constraints (both extensive and intentional) and agents, with their corresponding route and hosting costs.

Listing 6.1 shows the YAML definition of an extremely simple graph coloring problems with two variables and two agents.

8. Threads must compete for acquiring the Global Interpreter Lock, see https://wiki.python.org/moin/GlobalInterpreterLock
9. https://akka.io/
10. http://yaml.org

```yaml
name: graph coloring
objective: min

domains:
  colors:
    values: ['R', 'G']

variables:
  v1:
    domain: colors
  v2:
    domain: colors

constraints:
    pref_1:
      type: extensional
      variables: v1
      values:
        -0.1: R
        0.1: G
    diff_1_2:
      type: intention
      function: 10 if v1 == v2 else 0

agents: [a1, a2]
```

Listing 6.1 – Simple graph coloring in YAML

Additionally, pyDCOP also uses custom file formats for distribution, replication and scenario (for dynamic **DCOP**).

6.4.2 Command-line Interface

The main interface for pyDCOP is a command-line application, which can be used to solve a **DCOP**, distribute a computation graph, run replication or run a full dynamic **DCOP** (in which case a scenario thats defines events must be provided), etc.

pyDCOP can also be used through its API –although we will not cover this here– which should allow integrating it into other frameworks and experimentation libraries.

The full documentation for the command-line interface is available online [11] and is based on a *verb/action* paradigm. The main actions it supports are listed below:

- `pydcop solve` solves a **DCOP** using the thread runtime environment.
- `pydcop distribute` computes a distribution for a computation graph derived from a **DCOP**.
- `pydcop generate` generates benchmark problems: several types of graph coloring problems, distributed meetings scheduling and **SECP** are supported.

11. https://pydcop.readthedocs.io/en/latest/usage/cli_ref.html

- `pydcop run` continuously runs a dynamic **DCOP**, injecting events according to a predefined scenario.
- `pydcop agent` runs a single agent with the process runtime environment, which is used for example when deploying the systems on physically distributed computers.

6.4.3 Solving DCOP with pyDCOP

The `pydcop solve` command is the simplest way of solving a **DCOP** using pyDCOP, as it hides all complexities and defines defaults values for all settings. Using this command, solving a **DCOP** can be as simple as running:

```
$ pydcop solve --algo dpop graph_coloring.yaml
```

With this simple command, pyDCOP performs the following operations:

- starting threaded agents,
- computing a distribution,
- deploying the computations on the agents, according to the distribution,
- monitoring agents,
- collecting metrics and results and outputing them at the end of the solving process.

Of course, the `solve` command supports many options. For example, the following command solves a **DCOP** defined in `graph_coloring_50.yaml` using the synchronous **DSA** algorithm, for 30 cycles. Computations are distributed using the **GH-CGDP** heuristic and runtime metrics are collected at every cycle.

```
$ pydcop solve --distribution gh_cgdp --algo dsa  \
  --algo_params --algo_param stop_cycle:30 \
  --algo_param variant:C --algo_param probability:0.5 \
  --collect_on cycle_change --run_metric ./metrics.csv \
  graph_coloring_50.yaml
```

The metrics collected in `results.csv` allow analyzing the evolution of the solution quality during the solving process. Figure 6.6 is generated from such a metric file.

When needed, all the operations executed automatically by the `solve` command can also be run individually using their own specific command, please refer to the online documentation for these.

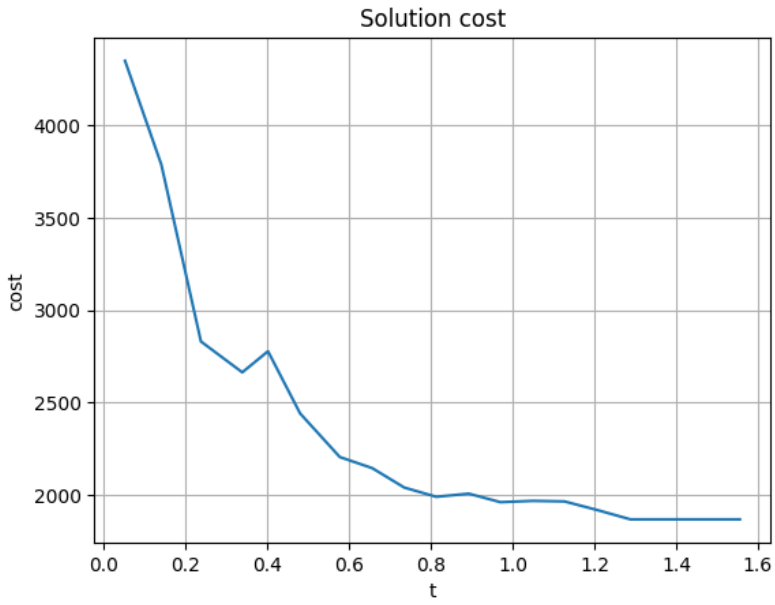


Figure 6.6 – Evolution of costs plotted from metrics output by the `pydcop solve` command

6.4.4 Programming with pyDCOP

pyDCOP provides many extension points to allow users to implement their own solution methods. New DCOP algorithm can of course be implemented, and distributed methods, replication, self-repair mechanisms as well.

Beside these extension points, many components of pyDCOP can also be replaced by new implementations that respect the same interface, like for example the communication or the directory service.

We are not going to cover all these possibilities in this document, instead we will simply demonstrate how to develop a simple DCOP algorithm using a simplified version of DSA.

Developing a new algorithm in pyDCOP simply amounts to creating a python module (i.e. a file) named after the algorithm. This module must define:

- the graphical model used by this algorithm: constraint graph, factor graph or DFS tree,
- the message(s) used by the algorithm,
- a class that derives from one of the `Computation` base classes, which represents the implementation of the computation for this algorithm.

Listing 6.2 shows a basic skeleton for such a module. As we are implementing a synchronous algorithm that defines computations for variables, the computation class is derived from `SynchronousComputationMixin` and `VariableComputation`. At each cycle, the `on_new_cycle()` method will be called with all the messages sent by neighbors in the previous cycle. Then, one simply has to implement this method using some of the many utility functions provided by pyDCOP to obtain a working implementation of DSA. Listing 6.3 provides an example of such implementation. Notice that the use of the utility functions provided by pyDCOP, like `assignment_cost()` and `find_optimal()`, helps keeping this implementation short and clear.

```
GRAPH_TYPE = 'constraints_hypergraph'

DsaMessage = message_type("dsa_value", ["value"])

class DsaTutoComputation(SynchronousComputationMixin,
VariableComputation):

    def __init__(self, computation_definition):
        ...

    def on_start(self):
        ...

    @register("dsa_value")
    def on_value_msg(self, variable_name, recv_msg, t):
        pass

    def on_new_cycle(self, messages, cycle_id) -> Optional[List]:
        ...
```

Listing 6.2 – Skeleton for a DCOP algorithm implementaion

```
def on_new_cycle(self, messages, cycle_id) -> Optional[List]:

    assignment = {self.variable.name: self.current_value}
    for sender, (message, t) in messages.items():
        assignment[sender] = message.value

    current_cost = assignment_cost(assignment, self.constraints)
    arg_min, min_cost = find_optimal(
        self.variable, assignment, self.constraints, self.mode
    )

    if current_cost - min_cost > 0 and 0.5 > random.random():
        self.value_selection(arg_min[0])

    self.post_to_all_neighbors(DsaMessage(self.current_value))
```

Listing 6.3 – Cycle handler for a simple DSA implementation

6.5 Sample Applications and Demonstration

All experimental evaluations presented in this thesis have been implemented using pyDCOP. pyDCOP can also be used to build physical distributed systems prototyping the application of **DCOP** in distributed and dynamic systems like **AmI**, **IoT**, edge computing and many other application domains.

pyDCOP has also been used to build a demonstration, showcased [125] at **JFSMA** in 2018. This demonstration illustrates a k-resilient distributed decision making process in an IoT system. Our scenario is based on a classical distributed weighted graph coloring problem, to which many real problems can be mapped. Each variable in the system maps to a vertex in the graph and can take one color as a value. Edges of the graph maps to binary constraints, assigning a cost for each combination of colors taken by its associated variables/vertices. The goal is to find a assignment of colors that minimize the sum of these costs.

Figure 6.7 – pyDCOP physical demonstrator

The demonstrator (see Figure 6.7) is made of a 3×3 grid of small single-board computers (Raspberry Pis), each fitted with a small touch-screen. Each of these computers runs one pyDCOP agent and displays a graphical interface presenting the current state of this agent. A central screen (an internet browser on a TV or computer screen) gives an overall view of the system and the current runtime metrics. During the demonstration, we dynamically remove random agents from the system. Remaining agents coordinate autonomously the repair process, which can be observed on their graphical interface. The self-repair, which implements **DRPM**[**DMCM**] (see Section 5.5) is also totally decentralized and is based on a distributed replication protocol followed by a host-selection mechanism modeled using a DCOP.

A video presenting this demonstration is available online [12].

12. urlhttps://www.dropbox.com/s/ozb0scwskkxqx6p/demoPyDCOP.mp4

7 Conclusion

In this concluding chapter, we first summarize the contributions presented in this dissertation. Then we give an overview of the dissemination activities concerning these contributions and present some paths for future research.

7.1 Summary of Contributions

In this dissertation, we investigated the usage of **Distributed Constraint Optimization Problem** to implement coordination among connected devices in **Ambient Intelligence** environments. We firmly believe that the centralized approaches used by currently deployed state-of the art solutions do not allow to reach the full potential of **AmI** and **IoT** applications in general. Therefore, we propose several contributions to field of **DCOP**, specifically designed to ease the use of this framework in real-life distributed **AmI** systems.

- First, we proposed in Chapter 3 a model for coordination among connected objects in a smart home. This model is goal-oriented: users do not need to care about complicated low-level details of home automation and can instead simply set *goals* to the smart environment. Then, based on the physical relations between the objects and the environment encoded in the model, devices autonomously and spontaneously find the way to reach these goals. We mapped this model to a **DCOP**, allowing us to leverage the large number of **DCOP** solution methods to implement cooperative coordination among the connected devices in the home. Through experimental evaluation, we showed that our model is viable approach for implementing coordination in such environments and identified **DCOP** solution methods best suited for these settings.
- Then, we investigated in Chapter 4 the distribution of decisions in these environments. We demonstrated that this subject is of paramount importance for applying **DCOP** approaches in real-world settings, even though it is seldom studied in the **DCOP** community. We proposed several distribution methods, both for the **SECP** model and for generic computation graphs in **IoT** systems. In both cases, we provided a definition of optimal distribution and presented an **ILP** to compute it. Experimentally, such optimal distributions proved to be too expensive

to obtain for all but the smallest systems. However, the greedy heuristics we proposed produce near-optimal results and are several orders of magnitude faster, which allows using them on realistic large systems.

- In Chapter 5 we studied the effects of dynamics in these systems. Smart homes, **IoT** and open multi-agent systems in general are open systems where agents can join and leave at any time and the definition of the problem itself can change over time, while devices are actively working on it. As a consequence, our model for coordination must be able to account for such changes if we want to apply it in real life. We discussed the most appropriate **DCOP** solution methods for dealing with changes at the problem level in **AmI** settings and proposed several approaches for handling agents' arrival and removal at run-time. We focused on agent failure, as this is the most problematic case, and devised methods for restoring a complete distribution of decision in such situations. Most notably, we introduced the concept of k-resilience, which characterizes a system able to survive the failure of up to k agents. To implement this concept in our **AmI** systems, we proposed **DRPM**[**DMCM**], a technique based on replication and distributed self-repair. Experimental evaluation of this method shows that it succeeds in repairing systems even when a large proportion of agents fails.
- While studying the aforementioned topics, we realized that no software library was available for studying the use of the **DCOP** framework in such distributed settings, especially for issues like distribution and resilience. Therefore we developed and open-sourced pyDCOP, introduced in Chapter 6, a software library designed for studying **DCOP** algorithms with a focus on challenges arising from their use in real-life systems, like distribution and fault-tolerance. pyDCOP makes it easier to develop new solution methods in this area and to prototype and test them on real devices. As its name suggest, pyDCOP is developed in python, which should make it accessible to a large population of researchers and students and allows its users to tap on the vast ecosystem of scientific libraries available in python. We hope that pyDCOP, will foster research on these topics.

7.2 Dissemination

The contributions presented in this dissertation have been published in articles presented at several scientific venues with peer reviewing, as listed here:

- The **SECP** model and its mapping to a **DCOP** have been presented at **IJCAI** and **JFSMA**:

  [128] P. Rust, G. Picard, and F. Ramparany. « Using Message-Passing DCOP Algorithms to Solve Energy-Efficient Smart Environment Configuration Problems ». In: *International Joint Conference on Artificial Intelligence*. IJCAI. IJCAI. 2016, p. 7

  [122] P. Rust, G. Picard, and F. Ramparany. « Approche DCOP pour résoudre des problèmes de configuration économe d'environnements intelligents ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2016

- The optimal and heuristic distribution methods for **SECP** presented in this dissertation are an extension of the works initially presented at **OptMAS** (best paper) and **JFSMA**:

[126] P. Rust, G. Picard, and F. Ramparany. « On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. vol. 10642. OPTMAS. 2017, pp. 116–137

[123] P. Rust, G. Picard, and F. Ramparany. « Déploiement de graphes de facteurs pour l'exécution d'algorithmes DCOP sur des infrastructures ouvertes ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2017

- Solution methods for the generalized distribution for **IoT** systems and repair techniques have been presented at **AAMAS**, **OptMAS** and **JFSMA**

[121] P. Rust, G. Picard, and F. Ramparany. « Self-Organized and Resilient Distribution of Decisions over Dynamic Multi-Agent Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. OPTMAS. 2018, p. 15

[124] P. Rust, G. Picard, and F. Ramparany. « Installing Resilience in Distributed Constraint Optimization Operated by Physical Multi-Agent Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. AAMAS. AAMAS. 2019, p. 3

[127] P. Rust, G. Picard, and F. Ramparany. « Résilience et auto-réparation de processus de décisions multi-agents ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2019

- The pyDCOP software library has been presented to the community at **OptMAS** and has been used for a physical demonstration at **JFSMA**:

[120] P. Rust, G. Picard, and F. Ramparany. « pyDCOP: A DCOP Library for Dynamic IoT Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. OPTMAS. 2019, p. 5

[125] P. Rust, G. Picard, and F. Ramparany. « Mise En Place d'une Décision Collective Résiliente Sur Une Infrastructure IoT à l'aide Du Framework PyDCOP (Démonstration) ». In: *Journées Francophones Sur Les Systèmes Multi-Agents*. JFSMA. JFSMA. 2018, pp. 223–224

Additionally, parts of these works and the pyDCOP library have been presented during tutorials at several occasions:

- **EASSS** in 2018
- **AAMAS** in 2018 and 2019
- **PFIA** in 2019

Several presentations of these works have also been made at Orange, including a presentation to the scientific board in 2017.

7.3 Paths for Future Research

Most of the topics covered by this thesis are open to further research, as follows.

DCOP Algorithms. We have only been using 'off the shelf' **DCOP** algorithms in our works, and simply tried to tune their parameters to optimize their behavior on our problems. However, we believe that better algorithms should be designed to handle the specificities of **AmI** and **IoT** systems. In many cases, it is probably possible to exploit the domain characteristics and the problem structure to improve the performance of current solution methods; for example, using domain pruning, decimation, edges cutting or many other techniques.

One specific aspect on which **DCOP** algorithms must progress is the handling of problems that have both soft and hard constraints. Current solution methods struggle on such problems, but we argue that most models for real-world problems, like **SECP** or distributed meeting scheduling, require it. For example, we believe that a variant of **MGM-2** could be developed for that purpose, as mentioned in Section 5.7.3, and would be an ideal candidate for implementing the repair phase in **DRPM**[**DMCM**]. Similarly, variants of **A-MaxSum** could also be tailored to the mix of hard and soft found in **SECP**.

Another aspect that requires further research is the development of asynchronous algorithms that support messages loss: very few current algorithms support this, even tough it is required for robust distributed systems.

SECP. Regarding the **SECP** model presented in this thesis, we assumed (as stated in Section 3.1.4) that physical models are known. In order to use the model in a real environment, it would be required to devise a technique to learn these models and adapt them at run-time, to represent the changes in the environment. Given the constrained devices these environments are made of, lightweight learning mechanisms should be used ; gaussian processes for example might be well suited for that task, as they are able to model accurately the behaviour of local phenomena.

The **SECP** model could also be improved on other aspects, for example it does not currently take into accounts the delay between and action and its effect on the environment: when switching a heater on for instance, one has to wait before the temperature increases in the room. In order to take into account such delay, some planning component should probably be embedded in the model.

Application to other domains. The techniques for distribution and resilience presented here have been designed for Ambient Intelligence. However, we believe that they could be re-used and adapted to other domains were a cooperative distributed decision making process is required. For example, these solution methods could be applied to the placement of workloads in distributed and cloud computing, especially when using edge-computing. Such placements depend on many criteria and, when scaling, these systems become to large to allow for a centralized solution; distributed localized solution methods could provide both the scalability and the robustness required in these environments. **Network Function Virtualizations** (**NFV**), which allow more flexible management and deployment of networks by telecommunication operators, must be placed on a physical infrastructure according to constraints and requirements on the resources and the network use, which may changer over time. This placement is another area where these methods could be used, as [111] already investigated, especially given that scalability and robustness are paramount in these applications

Implementations. Regarding the experimental evaluation and prototyping of distributed systems

based on the ideas presented in this thesis, and on Distributed Constraint Reasoning in general, further work is due on the implementations of framework, tools and libraries. The **DCOP** community has no repository with implementations of the many algorithms proposed over the years in this domain, not even for the algorithms that are considered to be *standard*. This state of affairs hinders research considerably and make comparisons and evaluation both difficult and error prone. We believe that such repository is required, along with better metrics and tools, to evaluate solution methods and better analyse their fitness to various application domains. pyDCOP is a first step in that direction, which should be pursued.

As a final conclusion, we want to highlight the fact that, while we only focused in this thesis on the use of **DCOP** for **AmI** and **IoT** systems, we believe that many other Artificial Intelligence approaches, besides the multi-agent area, should be studied to improve functionality and acceptability of these cyberphysical systems. *Semantic* approaches to improve expressivity of services and thing capabilities, *Machine Learning* techniques to adapt services and analyze complex socio-technical behaviors, *Context-acquisition and awareness* to install adapted and user-friendly behaviors, and *Deliberative multi-agent architectures* to install sustainable and explainable interactions with users, are just a few examples of topics that are currently studied and which could bring interesting advances for these applications domains.

Bibliography

[1] M. Abadi et al. « TensorFlow: A System for Large-Scale Machine Learning ». In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI. OSDI'16. May 27, 2016, pp. 265–283. arXiv: 1605.08695.

[2] M. Abdoos, N. Mozayani, and A. L. Bazzan. « Holonic Multi-Agent System for Traffic Signals Control ». In: *Engineering Applications of Artificial Intelligence* 26.5-6 (May 2013), pp. 1575–1587. ISSN: 09521976.

[3] W. P. Adams, R. J. Forrester, and F. W. Glover. « Comparisons and Enhancement Strategies for Linearizing Mixed 0-1 Quadratic Programs ». In: *Discrete Optimization* 1.2 (Nov. 2004), pp. 99–120. ISSN: 15725286.

[4] agent-software. *JACK*. 2019.

[5] M. R. Alam, M. B. I. Reaz, and M. A. M. Ali. « A Review of Smart Homes—Past, Present, and Future ». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (Nov. 2012), pp. 1190–1203. ISSN: 1094-6977, 1558-2442.

[6] E. Amouroux, T.-Q. Chu, A. Boucher, and A. Drogoul. « GAMA: An Environment for Implementing and Running Spatially Explicit Multi-Agent Simulations ». In: *Agent Computing and Multi-Agent Systems* (Berlin, Heidelberg). Ed. by A. Ghose, G. Governatori, and R. Sadananda. 2009, pp. 359–371.

[7] J. C. Augusto. « Ambient Intelligence: The Confluence of Ubiquitous/Pervasive Computing and Artificial Intelligence ». In: *Intelligent Computing Everywhere*. 2007, pp. 213–234.

[8] A. Barabasi and R. Albert. « Emergence of Scaling in Random Networks ». In: *Science* 286.5439 (Oct. 15, 1999), pp. 509–512. ISSN: 00368075, 10959203.

[9] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. 2007. 286 pp.

[10] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. « The Complexity of Decentralized Control of Markov Decision Processes ». In: *Mathematics of Operations Research* 27.4 (Nov. 2002), pp. 819–840. ISSN: 0364-765X, 1526-5471.

[11] C.-E. Bichot and P. Siarry, eds. *Graph Partitioning*. OCLC: 765366069. 2011. 368 pp.

[12] K. Binmore. *Fun and Games: A Text on Game Theory*. 2. pr. OCLC: 845425112. 1992. 602 pp.

[13] J. Boes, F. Migeon, P. Glize, and E. Salvy. « Model-Free Optimization of an Engine Control Unit Thanks to Self-Adaptive Multi-Agent Systems ». In: *International Conference on Embedded Real Time Software and Systems*. ERTS. 2014, p. 12.

[14] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. « Multi-Agent Oriented Programming with JaCaMo ». In: *Science of Computer Programming* 78.6 (June 2013), pp. 747–761. ISSN: 01676423.

[15] R. H. Bordini, J. F. Hübner, and M. J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. OCLC: 255545467. 2007. 273 pp.

[16] M. Boulle. « Compact Mathematical Formulation for Graph Partitioning ». In: *Optimization and Engineering* 5.3 (Sept. 2004), pp. 315–333. ISSN: 1389-4420.

[17] F. Bousquet, I. Bakam, H. Proton, and C. Le Page. « Cormas: Common-Pool Resources and Multi-Agent Systems ». In: *Tasks and Methods in Applied Artificial Intelligence* (Berlin, Heidelberg). Ed. by A. Pasqual del Pobil, J. Mira, and M. Ali. 1998, pp. 826–837.

[18] M. Bratman. *Intention, Plans, and Practical Reason*. David Hume Series. 1999. 200 pp.

[19] M. Burger, G. Notarstefano, F. Allgower, and F. Bullo. « A Distributed Simplex Algorithm and the Multi-Agent Assignment Problem ». In: *Proceedings of the 2011 American Control Conference*. 2011 American Control Conference. June 2011, pp. 2639–2644.

[20] J. Cerquides, R. Emonet, G. Picard, and J. A. Rodriguez-Aguilar. « Decimaxsum: Using Decimation to Improve Max-Sum on Cyclic Dcops ». In: *21st International Conference of the Catalan Association for Artificial Intelligence (CCIA 2018)* (Roses, Spain). Vol. 308. Artificial Intelligence Research and Development - Current Challenges, New Trends and Applications. Oct. 2018, pp. 27–36.

[21] A. Chechetka and K. Sycara. « No-Commitment Branch and Bound Search for Distributed Constraint Optimization ». In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems - AAMAS '06*. AAMAS. 2006, p. 1427.

[22] T.-Y. Cheung. « Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation ». In: *IEEE Transactions on Software Engineering* SE-9.4 (July 1983), pp. 504–512. ISSN: 0098-5589.

[23] L. Cohen and R. Zivan. « Max-Sum Revisited: The Real Power of Damping ». In: *Autonomous Agents and Multiagent Systems*. Ed. by G. Sukthankar and J. A. Rodriguez-Aguilar. Vol. 10643. 2017, pp. 111–124.

[24] Z. Collin and S. Dolev. « Self-Stabilizing Depth-First Search ». In: *Information Processing Letters* 49.6 (Mar. 22, 1994), pp. 297–301. ISSN: 0020-0190.

[25] A. Components. *Jadex - Active Components*. 2019.

[26] F. Cruz, P. Gutierrez, and P. Meseguer. « Simulation vs Real Execution in DCOP Solving ». In: *OPTAMAS*. AAMAS. 2014, p. 12.

[27] L. C. De Silva, C. Morikawa, and I. M. Petra. « State of the Art of Smart Homes ». In: *Engineering Applications of Artificial Intelligence* 25.7 (Oct. 2012), pp. 1313–1321. ISSN: 09521976.

[28] R. Dechter. *Constraint Processing*. 2003. 481 pp.

[29] R. Dechter and J. Pearl. « Tree Clustering for Constraint Networks ». In: *Artificial Intelligence* 38.3 (Apr. 1989), pp. 353–366. ISSN: 00043702.

[30] V. Degeler and A. Lazovik. « Dynamic Constraint Reasoning in Smart Environments ». In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 2013 IEEE

25th International Conference on Tools with Artificial Intelligence (ICTAI). Nov. 2013, pp. 167–174.

[31] P. J. Denning, ed. *The Invisible Future: The Seamless Integration of Technology into Everyday Life*. 2002.

[32] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. « Self-Organization in Multi-Agent Systems ». In: *The Knowledge Engineering Review* 20.2 (June 2005), pp. 165–189. ISSN: 0269-8889, 1469-8005.

[33] S. Dolev. *Self-Stabilization*. OCLC: 957241720. 2000.

[34] N. Fan and P. M. Pardalos. « Linear and Quadratic Programming Approaches for the General Graph Partitioning Problem ». In: *Journal of Global Optimization* 48.1 (Sept. 2010), pp. 57–71. ISSN: 0925-5001, 1573-2916.

[35] A. Farinelli, A. Rogers, and N. R. Jennings. « Agent-Based Decentralised Coordination for Sensor Networks Using the Max-Sum Algorithm ». In: *Autonomous Agents and Multi-Agent Systems* 28.3 (May 2014), pp. 337–380. ISSN: 1387-2532, 1573-7454.

[36] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. « Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm ». In: *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*. AAMAS 2008. 2008, pp. 639–646.

[37] F. Fioretto, T. Le, W. Yeoh, E. Pontelli, and T. C. Son. « Improving DPOP with Branch Consistency for Solving Distributed Constraint Optimization Problems ». In: *Principles and Practice of Constraint Programming* (Cham). Ed. by B. O'Sullivan. 2014, pp. 307–323.

[38] F. Fioretto, E. Pontelli, and W. Yeoh. « Distributed Constraint Optimization Problems and Applications: A Survey ». In: *Journal of Artificial Intelligence Research* 61 (Mar. 29, 2018), pp. 623–698. ISSN: 1076-9757. arXiv: 1602.06347.

[39] F. Fioretto and W. Y. E. Pontelli. « A Multiagent System Approach to Scheduling Devices in Smart Homes ». In: *Proceedings of the International Workshop on Artificial Intelligence for Smart Grids and Smart Buildings*. Workshop on Artificial Intelligence for Smart Grids and Smart Buildings. 2017, p. 7.

[40] S. Fitzpatrick and L. Meertens. « Distributed Coordination through Anarchic Optimization ». In: *Distributed Sensor Networks*. Ed. by V. Lesser, C. L. Ortiz, and M. Tambe. Red. by G. Weiss. Vol. 9. 2003, pp. 257–295.

[41] F. D. Fomeni and A. N. Letchford. « A Dynamic Programming Heuristic for the Quadratic Knapsack Problem ». In: *INFORMS Journal on Computing* 26.1 (Feb. 2014), pp. 173–182. ISSN: 1091-9856, 1526-5528.

[42] gartner. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. 2017.

[43] A. Gershman, A. Meisels, and R. Zivan. « Asynchronous Forward Bounding for Distributed COPs ». In: *Journal of Artificial Intelligence Research* 34 (Feb. 10, 2009), pp. 61–88. ISSN: 1076-9757.

[44] A. K. Gopalakrishna, T. Ozcelebi, A. Liotta, and J. J. Lukkien. « Exploiting Machine Learning for Intelligent Room Lighting Applications ». In: *2012 6th IEEE INTERNATIONAL*

CONFERENCE INTELLIGENT SYSTEMS. 2012 6th IEEE International Conference Intelligent Systems (IS). Sept. 2012, pp. 406–411.

[45] R. Greenstadt, B. Grosz, and M. D. Smith. « SSDPOP: Improving the Privacy of DCOP with Secret Sharing ». In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems - AAMAS '07*. The 6th International Joint Conference. 2007, p. 1.

[46] V. Guivarch. « Prise en compte de la dynamique du contexte pour les systèmes ambiantspar systèmes multi-agents adaptatifs ». Université Toulouse III -Paul Sabatier, 2014.

[47] O. Gutknecht and J. Ferber. « The MadKit Agent Platform Architecture ». In: *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*. Ed. by T. Wagner and O. F. Rana. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 1887. 2001, pp. 48–55.

[48] C. Hewitt, P. Bishop, and R. Steiger. « A Universal Modular ACTOR Formalism for Artificial Intelligence ». In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA). IJCAI'73. 1973, pp. 235–245.

[49] K. Hirayama and M. Yokoo. « Distributed Partial Constraint Satisfaction Problem ». In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. CP. Vol. 1330. 1997, pp. 222–236.

[50] K. D. Hoang, F. Fioretto, P. Hou, M. Yokoo, W. Yeoh, and R. Zivan. « Proactive Dynamic Distributed Constraint Optimization ». In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. 2016, pp. 597–605.

[51] K. D. Hoang, P. Hou, F. Fioretto, W. Yeoh, R. Zivan, and M. Yokoo. « Infinite-Horizon Proactive Dynamic DCOPs ». In: (2017), p. 9.

[52] B. Horling and V. Lesser. « A Survey of Multi-Agent Organizational Paradigms ». In: *The Knowledge Engineering Review* 19.04 (2004), pp. 281–316.

[53] J. F. Hübner, O. Boissier, R. Kitio, and A. Ricci. « Instrumenting Multi-Agent Organisations with Organisational Artifacts and Agents: "Giving the Organisational Power Back to the Agents" ». In: *Autonomous Agents and Multi-Agent Systems* 20.3 (May 2010), pp. 369–400. ISSN: 1387-2532, 1573-7454.

[54] J. F. Hubner and J. S. Sichman. « SACI: Uma Ferramenta para Implementação e Monitoração da Comunicação entre Agentes ». In: (2000), p. 12.

[55] IETF. *CoAP: Constrained Application Protocol*. ISBN 2070-1721. 2014.

[56] IETF. *CoRE Resource Directory*. 2019.

[57] IETF. *DNS-SD*. 2013.

[58] IETF. *mDNS*. 2013.

[59] G. Kahn. « The Semantics of a Simple Language for Parallel Programming ». In: (), p. 6.

[60] E. Kaldeli, E. U. Warriach, A. Lazovik, and M. Aiello. « Coordinating the Web of Services for a Smart Home ». In: *ACM Transactions on the Web* 7.2 (May 1, 2013), pp. 1–40. ISSN: 15591131.

[61] R. M. Karp and R. E. Miller. « Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing ». In: *SIAM Journal on Applied Mathematics* 14.6 (Nov. 1966), pp. 1390–1411. ISSN: 0036-1399, 1095-712X.

[62] M. Khan, L. Tran-Thanh, W. Yeoh, and N. R. Jennings. « A Near-Optimal Node-to-Agent Mapping Heuristic for GDL-Based DCOP Algorithms in Multi-Agent Systems ». In: *Roceedings of the International Conference on Autonomous Agents and Multiagent Systems*. AAMAS. 2018, pp. 1613–1621.

[63] C. Kiekintveld, Z. Yin, A. Kumar, and M. Tambe. « Asynchronous Algorithms for Approximate Distributed Constraint Optimization with Quality Bounds ». In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1-Volume 1*. 2010, pp. 133–140.

[64] W. Kluegel, M. A. Iqbal, F. Fioretto, W. Yeoh, and E. Pontelli. « A Realistic Dataset for the Smart Home Device Scheduling Problem for DCOPs ». In: *Autonomous Agents and Multiagent Systems*. OPTMAS. Vol. 10643. 2017, pp. 125–142.

[65] F. R. Kschischang and B. J. Frey. « Iterative Decoding of Compound Codes by Probability Propagation in Graphical Models ». In: *IEEE Journal on Selected Areas in Communications* 16.2 (Feb. 1998), pp. 219–230. ISSN: 0733-8716.

[66] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. « Factor Graphs and the Sum-Product Algorithm ». In: (2001), p. 76.

[67] Y. Kubera, P. Mathieu, and S. Picault. « IODA: An Interaction-Oriented Approach for Multi-Agent Based Simulations ». In: *Autonomous Agents and Multi-Agent Systems* 23.3 (Nov. 2011), pp. 303–343. ISSN: 1387-2532, 1573-7454.

[68] A. Kumar, A. Petcu, and B. Faltings. « H-DPOP: Using Hard Constraints for Search Space Pruning in DCOP ». In: *Proceedings of the Twenty-Third {AAAI} Conference on Artificial Intelligence, {AAAI} 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI. 2008, pp. 325–330.

[69] R. N. Lass, E. Sultanik, and W. C. Regli. « Dynamic Distributed Constraint Reasoning. » In: *AAAI*. 2008, pp. 1466–1469.

[70] T. Léauté, B. Ottens, R. Szymanek, and É. P. F. De. « FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization ». In: *In Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09*. IJCAI-DCR. 2009, pp. 160–164.

[71] C. Lee, D. Nordstedt, and S. Helal. « Enabling Smart Spaces with OSGi ». In: *IEEE Pervasive Computing* 2.3 (July 2003), pp. 89–94. ISSN: 1536-1268.

[72] E. Lee and D. Messerschmitt. « Synchronous Data Flow ». In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219.

[73] C. J. van Leeuwen and P. Pawelczak. « CoCoA: A Non-Iterative Approach to a Local Search (A)DCOP Solver ». In: *AAAI Conference on Artficial Intelligence*. Feb. 2017.

[74] B. Lutati, I. Gontmakher, M. Lando, A. Netzer, A. Meisels, and A. Grubshtein. « AgentZero: A Framework for Simulating and Evaluating Multi-Agent Algorithms ». In: *Agent-Oriented Software Engineering*. 2014, pp. 309–327.

[75] N. A. Lynch. *Distributed Algorithms*. OCLC: 1047802139. 1997.

[76] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth. « Machine Learning for Internet of Things Data Analysis: A Survey ». In: *Digital Communications and Networks* 4.3 (Aug. 2018), pp. 161–175. ISSN: 23528648.

[77] R. T. Maheswaran, J. P. Pearce, and M. Tambe. « Distributed Algorithms for DCOP: A Graphical-Game-Based Approach. » In: *ISCA PDCS*. 2004, pp. 432–439.

[78] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. « Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling ». In: *AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS. 2004, p. 8.

[79] R. Mailler and V. Lesser. « Solving Distributed Constraint Optimization Problems Using Cooperative Mediation ». In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '04. 2004, pp. 438–445.

[80] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. « Pregel: A System for Large-Scale Graph Processing ». In: *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. 2009, p. 1.

[81] A. F. T. Martins, M. A. T. Figueiredo, P. M. Q. Aguiar, N. A. Smith, and E. P. Xing. « AD3: Alternating Directions Dual Decomposition for MAP Inference in Graphical Models ». In: *The Journal of Machine Learning Research* (2015), p. 51.

[82] T. Matsui, H. Matsuo, and M. Ca. « A Quantified Distributed Constraint Optimization Problem ». In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems:* AAMAS. Vol. 1. 2010, p. 9.

[83] S. Mazac, F. Armetta, and S. Hassas. « On Bootstrapping Sensori-Motor Patterns for a Constructivist Learning System in Continuous Environments ». In: *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*. ALIFE 14. July 30, 2014, pp. 160–167.

[84] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. « Comparing Performance of Distributed Constraints Processing Algorithms ». In: *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*. 2002, pp. 86–93.

[85] J. E. Mitchell. « Branch-and-Cut Algorithms for Combinatorial Optimization Problems ». In: *Handbook of Applied Optimization*. 2000, p. 19.

[86] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. « Adopt: Asynchronous Distributed Constraint Optimization with Quality Guarantees ». In: *Artificial Intelligence* 161.1-2 (Jan. 2005), pp. 149–180. ISSN: 00043702.

[87] H. Moens and F. D. Turck. « VNF-P: A Model for Efficient Placement of Virtualized Network Functions ». In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. 2014 10th International Conference on Network and Service Management (CNSM). Nov. 2014, pp. 418–423.

[88] Y. Naveh, R. Zivan, and W. Yeoh. « Resilient Distributed Constraint Optimization Problems ». In: OPTMAS. 2017, p. 14.

[89] A. Netzer, A. Grubshtein, and A. Meisels. « Concurrent Forward Bounding for Distributed Constraint Optimization Problems ». In: *Artificial Intelligence* 193 (Dec. 2012), pp. 186–216. ISSN: 00043702.

[90] D. T. Nguyen, W. Yeoh, and H. C. Lau. « Distributed Gibbs: A Memory-Bounded Sampling-Based DCOP Algorithm ». In: *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*. AAMAS. 2013, pp. 167–174.

[91] S. Okamoto, R. Zivan, and A. Nahon. « Distributed Breakout: Beyond Satisfaction ». In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI. 2016, pp. 447–453.

[92] osgi. *OSGi™ Alliance – The Dynamic Module System for Java*.

[93] B. Ottens, C. Dimitrakakis, and B. Faltings. « DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimization Problems ». In: *Proceedings of AAAI*. AAAI. Vol. 8. 2012, pp. 528–534.

[94] M. T. Özsu and P. Valduriez. « Data Replication ». In: *Principles of Distributed Database Systems, Third Edition*. 2011, pp. 459–495.

[95] J. Palanca, E. del Val, A. Garcia-Fornes, H. Billhardt, J. M. Corchado, and V. Julián. « Designing a Goal-Oriented Smart-Home Environment ». In: *Information Systems Frontiers* 20.1 (Feb. 2018), pp. 125–142. ISSN: 1387-3326, 1572-9419.

[96] C. Parra, D. Romero, S. Mosser, R. Rouvoy, L. Duchien, and L. Seinturier. « Using Constraint-Based Optimization and Variability to Support Continuous Self-Adaptation ». In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*. The 27th Annual ACM Symposium. 2012, p. 486.

[97] H. V. D. Parunak. « Industrial and Practical Applications of DAI ». In: *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Ed. by G. Weiss. 1999, pp. 377–421.

[98] J. P. Pearce and M. Tambe. « Quality Guarantees on K-Optimal Solutions for Distributed Constraint Optimization Problems ». In: *Proceedings of the International Joint Conference on Artificial Intelligence*. IJCAI. 2007, p. 1446–1451.

[99] F. Pecora and A. Cesta. « DCOP for Smart Homes: A Case Study ». In: *Computational Intelligence* 23.4 (Dec. 12, 2007), pp. 395–419. ISSN: 08247935.

[100] F. Pecora, P. Modi, and P. Scerri. « Reasoning about and Dynamically Posting N-Ary Constraints in ADOPT ». In: *7th International Workshop on Distributed Constraint Reasoning, at AAMAS*. Vol. 2006. 2006, p. 15.

[101] O. Peri and A. Meisels. « Synchronizing for Performance DCOP Algorithms ». In: *Proceedings of the 5th International Conference on Agents and Artificial Intelligence*. Icaart13. Vol. 1. 2013.

[102] A. Petcu. « ODPOP: An Algorithm for Open/Distributed Constraint Optimization ». In: *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, {USA}*. AAAI. 2006, pp. 703–708.

[103] A. Petcu. « PC-DPOP: A New Partial Centralization Algorithm for Distributed Optimization ». In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI. IJCAI'07. 2007, pp. 167–172.

[104] A. Petcu and B. Faltings. « A Distributed, Complete Method for Multi-Agent Constraint Optimization ». In: *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004)*. CP. 2004, p. 15.

[105] A. Petcu and B. Faltings. « Approximations in Distributed Optimization ». In: *Principles and Practice of Constraint Programming - CP 2005*. Vol. 3709. 2005, pp. 802–806.

[106] A. Petcu and B. Faltings. « Optimal Solution Stability in Dynamic, Distributed Constraint Optimization ». In: *2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07)*. 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07). Nov. 2007, pp. 321–327.

[107] A. Petcu and B. Faltings. « Superstabilizing, Fault-Containing Distributed Combinatorial Optimization. » In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 20. 2005, p. 449.

[108] F. Piette, C. Caval, C. Dinont, A. E. F. Seghrouchni, and P. Tailliert. « A Multi-Agent Solution for the Deployment of Distributed Applications in Ambient Systems ». In: *Engineering Multi-Agent Systems* (Cham). Ed. by M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern. 2016, pp. 156–175.

[109] V. Plantevin, A. Bouzouane, B. Bouchard, and S. Gaboury. « Towards a More Reliable and Scalable Architecture for Smart Home Environments ». In: *Journal of Ambient Intelligence and Humanized Computing* (Aug. 9, 2018). ISSN: 1868-5137, 1868-5145.

[110] H. K. Pung, T. Gu, and D. Q. Zhang. « Toward an OSGi-Based Infrastructure for Context-Aware Applications ». In: *IEEE Pervasive Computing* 3.4 (Oct. 2004), pp. 66–74. ISSN: 1536-1268.

[111] P. T. A. Quang, A. Bradai, K. D. Singh, G. Picard, and R. Riggio. « Single and Multi-Domain Adaptive Allocation Algorithms for VNF Forwarding Graph Embedding ». In: *IEEE Transactions on Network and Service Management* 16.1 (Mar. 2019), pp. 98–112. ISSN: 1932-4537, 2373-7379.

[112] H. Raiffa. *Decision Analysis; Introductory Lectures on Choices under Uncertainty*. OCLC: 449943. 1968.

[113] A. S. Rao and M. P. Georgeff. « BDI Agents: From Theory to Practice ». In: (1995), p. 8.

[114] A. Ricci, M. Piunti, and M. Viroli. « Environment Programming in Multi-Agent Systems: An Artifact-Based Perspective ». In: *Autonomous Agents and Multi-Agent Systems* 23.2 (Sept. 2011), pp. 158–192. ISSN: 1387-2532, 1573-7454.

[115] Rina Dechter and Avi Dechter. « Belief Maintenance in Dynamic Constraint Networks ». In: *Proceedings of the 7th National Conference on Artificial Intelligence*. 1988.

[116] S. Rodríguez, J. F. De Paz, G. Villarrubia, C. Zato, J. Bajo, and J. M. Corchado. « Multi-Agent Information Fusion System to Manage Data from a WSN in a Residential Home ». In: *Information Fusion* 23 (May 1, 2015), pp. 43–57. ISSN: 1566-2535.

[117] S. Rodriguez, N. Gaud, and S. Galland. « SARL: A General-Purpose Agent-Oriented Programming Language ». In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). Aug. 2014, pp. 103–110.

[118] A. Rogers, A. Farinelli, R. Stranders, and N. Jennings. « Bounded Approximate Decentralised Coordination via the Max-Sum Algorithm ». In: *Artificial Intelligence* 175.2 (Feb. 2011), pp. 730–759. ISSN: 00043702.

[119] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. In collab. with E. Davis and D. Edwards. Third edition, Global edition. Prentice Hall Series in Artificial Intelligence. OCLC: 945899984. 2016. 1132 pp.

[120] P. Rust, G. Picard, and F. Ramparany. « pyDCOP: A DCOP Library for Dynamic IoT Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. OPTMAS. 2019, p. 5.

[121] P. Rust, G. Picard, and F. Ramparany. « Self-Organized and Resilient Distribution of Decisions over Dynamic Multi-Agent Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. OPTMAS. 2018, p. 15.

[122] P. Rust, G. Picard, and F. Ramparany. « Approche DCOP pour résoudre des problèmes de configuration économe d'environnements intelligents ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2016.

[123] P. Rust, G. Picard, and F. Ramparany. « Déploiement de graphes de facteurs pour l'exécution d'algorithmes DCOP sur des infrastructures ouvertes ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2017.

[124] P. Rust, G. Picard, and F. Ramparany. « Installing Resilience in Distributed Constraint Optimization Operated by Physical Multi-Agent Systems ». In: *International Conference on Autonomous Agents and Multiagent Systems*. AAMAS. AAMAS. 2019, p. 3.

[125] P. Rust, G. Picard, and F. Ramparany. « Mise En Place d'une Décision Collective Résiliente Sur Une Infrastructure IoT à l'aide Du Framework PyDCOP (Démonstration) ». In: *Journées Francophones Sur Les Systèmes Multi-Agents*. JFSMA. JFSMA. 2018, pp. 223–224.

[126] P. Rust, G. Picard, and F. Ramparany. « On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments ». In: *International Conference on Autonomous Agents and Multiagent Systems*. OptMAS. Vol. 10642. OPTMAS. 2017, pp. 116–137.

[127] P. Rust, G. Picard, and F. Ramparany. « Résilience et auto-réparation de processus de décisions multi-agents ». In: *Journées Francophones sur les Systèmes Multi-Agents*. JFSMA. JFSMA. 2019.

[128] P. Rust, G. Picard, and F. Ramparany. « Using Message-Passing DCOP Algorithms to Solve Energy-Efficient Smart Environment Configuration Problems ». In: *International Joint Conference on Artificial Intelligence*. IJCAI. IJCAI. 2016, p. 7.

[129] F. Sadri. « Ambient Intelligence: A Survey ». In: *ACM Computing Surveys* 43.4 (Oct. 1, 2011), pp. 1–66. ISSN: 03600300.

[130] D. Saha and A. Mukherjee. « Pervasive Computing: A Paradigm for the 21st Century ». In: *Computer* 36.3 (Mar. 2003), pp. 25–31. ISSN: 0018-9162.

[131] T. Saraç and A. Sipahioglu. « Generalized Quadratic Multiple Knapsack Problem and Two Solution Approaches ». In: *Computers & Operations Research* 43 (Mar. 2014), pp. 78–89. ISSN: 03050548.

[132] « Self-Organising Systems ». In: *Self-Organising Software: From Natural to Artificial Adaptation*. Ed. by G. D. M. Serugendo, M.-P. Gleizes, and A. Karageorgos. Natural Computing Series. 2011, pp. 7–32.

[133] Y. Shoham. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. 2009.

[134] B. M. Smith. « Chapter 11 - Modelling ». In: *Foundations of Artificial Intelligence*. Ed. by F. Rossi, P. van Beek, and T. Walsh. Vol. 2. Handbook of Constraint Programming. Jan. 1, 2006, pp. 377–406.

[135] H. Song, S. Barrett, A. Clarke, and S. Clarke. « Self-Adaptation with End-User Preferences: Using Run-Time Models and Constraint Solving ». In: *Model-Driven Engineering Languages and Systems*. Vol. 8107. 2013, pp. 555–571.

[136] A. Stimson. *Photometry and Radiometry for Engineers*. OCLC: 833226111. 1974. 446 pp.

[137] E. A. Sultanik, R. N. Lass, and W. C. Regli. « DCOPolis: A Framework for Simulating and Deploying Distributed Constraint Reasoning Algorithms (Demo Paper) ». In: 2008, p. 2.

[138] Q. Sun, W. Yu, N. Kochurov, Q. Hao, and F. Hu. « A Multi-Agent-Based Intelligent Sensor and Actuator Network Design for Smart House and Home Automation ». In: *Journal of Sensor and Actuator Networks* 2.3 (Aug. 19, 2013), pp. 557–588. ISSN: 2224-2708.

[139] D. Tarlow, I. E. Givoni, and R. S. Zemel. « HOP-MAP: Efficient Message Passing with High Order Potentials ». In: *AISTATS*. AISTATS. 2010, p. 8.

[140] S. Valero, E. del Val, J. Alemany, and V. Botti. « Using Magentix2 in Smart-Home Environments ». In: *10th International Conference on Soft Computing Models in Industrial and Environmental Applications*. Ed. by Á. Herrero, J. Sedano, B. Baruque, H. Quintián, and E. Corchado. Vol. 368. 2015, pp. 27–37.

[141] L. G. Valiant. « A Bridging Model for Parallel Computation ». In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782.

[142] M. Vallée, F. Ramparany, and L. Vercouter. « A Multi-Agent System for Dynamic Service Composition in Ambient Intelligence Environments ». In: *Advances in Pervasive Computing, Adjunct Proceedings of the Third International Conference on Pervasive Computing*. Pervasive. 2005, p. 8.

[143] G. Verfaillie and N. Jussien. « Constraint Solving in Uncertain and Dynamic Environments: A Survey ». In: *Constraints* 10.3 (July 2005), pp. 253–281. ISSN: 1383-7133, 1572-9354.

[144] M. Vinyals, J. A. Rodriguez-Aguilar, and J. Cerquides. « Constructing a Unifying Theory of Dynamic Programming DCOP Algorithms via the Generalized Distributive Law ». In: *Autonomous Agents and Multi-Agent Systems* 22.3 (May 2011), pp. 439–464. ISSN: 1387-2532, 1573-7454.

[145] M. Wahbi, R. Ezzahir, C. Bessiere, and E. H. Bouyakhf. « DisChoco 2: A Platform for Distributed Constraint Reasoning ». In: *Proceedings of the IJCAI'11 Workshop on Distributed Constraint Reasoning* (Barcelona, Catalonia, Spain). DCR'11. 2011, pp. 112–121.

[146] M. Weiser. « The Computer for the 21th Century ». In: *Scientific American* 265.3 (1991), pp. 94–104.

[147] U. Wilensky. *NetLogo*. 1999.

[148] C.-L. Wu, C.-F. Liao, and L.-C. Fu. « Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology ». In: *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)* 37.2 (Mar. 2007), pp. 193–205. ISSN: 1094-6977.

[149] Xiaojing Ye and Junwei Huang. « A Framework for Cloud-Based Smart Home ». In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. 2011 International Conference on Computer Science and Network Technology (ICCSNT). Dec. 2011, pp. 894–897.

[150] E. P. Xing et al. « Petuum: A New Platform for Distributed Machine Learning on Big Data ». In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*. The 21th ACM SIGKDD International Conference. 2015, pp. 1335–1344.

[151] B. Yao, X. Liu, W.-J. Zhang, X.-E. Chen, X.-M. Zhang, M. Yao, and Z.-X. Zhao. « Applying Graph Theory to the Internet of Things ». In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. 2013 IEEE International Conference on High Performance Computing and Communications (HPCC) & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (EUC). Nov. 2013, pp. 2354–2361.

[152] W. Yeoh, P. Varakantham, X. Sun, and S. Koenig. « Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems ». In: *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 2015 IEEE / WIC / ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). Dec. 2015, pp. 257–264.

[153] M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. « The Distributed Constraint Satisfaction Problem: Formalization and Algorithms ». In: *IEEE Transactions on Knowledge and Data Engineering* 10.5 (Sept.-Oct./1998), pp. 673–685. ISSN: 10414347.

[154] M. Yokoo, T. Ishida, E. Durfee, and K. Kuwabara. « Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving ». In: *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. [1992] 12th International Conference on Distributed Computing Systems. 1992, pp. 614–621.

[155] M. Yokoo and K. Hirayama. « Distributed Breakout Algorithm for Solving Distributed Constraints Satisfaction Problems ». In: *Proceedings of the Second International Conference on Multiagent Systems*. International Conference on Multiagent Systems. 1996.

[156] M. Yokoo and K. Hirayama. « Algorithms for Distributed Constraint Satisfaction: A Review ». In: *Autonomous Agents and Multi-Agent Systems* (2000), p. 23.

[157] W. Zhang, G. Wang, and L. Wittenburg. « Distributed Stochastic Search for Constraint Satisfaction and Optimization: Parallelism, Phase Transitions and Performance ». In: *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*. 2002.

[158] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. « Distributed Stochastic Search and Distributed Breakout: Properties, Comparison and Applications to Constraint Optimization Problems in Sensor Networks ». In: *Artificial Intelligence* 161.1-2 (Jan. 2005), pp. 55–87. ISSN: 00043702.

[159] R. Zivan and A. Meisels. « Message Delay and DisCSP Search Algorithms ». In: *Annals of Mathematics and Artificial Intelligence* 46.4 (Oct. 27, 2006), pp. 415–439. ISSN: 1012-2443, 1573-7470.

[160] R. Zivan and H. Peled. « Max/Min-Sum Distributed Constraint Optimization through Value Propagation on an Alternating Dag ». In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. 2012, pp. 265–272.

Appendices

A Notations

Constraint Reasoning

Symbol	Description	Definition
$\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \mu \rangle$	DCOP	2.3.2
$\mathcal{X}$	Set of variables	2.3.2
x_i	One variable, $x_i \in \mathcal{X}$	2.3.2
$\mathcal{D}$	Set of domains for the variables	2.3.2
$\mathcal{D}_{x_i}$	Domain of variable x_i, $\mathcal{D}_{x_i} \in \mathcal{D}$	2.3.2
$\mathbf{x}_i^k$	A possible state of variable x_i, $\mathbf{x}_i^k \in \mathcal{D}_{x_i}$	2.3.2
$\mathcal{F}$	Set of constraints	2.3.2
f_j	A constraint $f_j \in \mathcal{F}$	2.3.2
S_{f_j}	Scope of constraint f_j	2.3.2
$\mathcal{A}$	Set of agents	2.3.2
a_a	Agent $a_a \in \mathcal{A}$	2.3.2
σ	An assignment of value to variables	2.3.2
μ	Mapping of variables to agents $\mu : \mathcal{X} \mapsto \mathcal{A}$	4.1.1
$\mu(x)$	Agent hosting a variable: $\mu(x) = a_a$	4.1.1
$\mu^{-1}(x)$	Set of variables hosted by an agent: $\mu^{-1}(x) \subseteq \mathcal{X}$	4.1.1

SECP

Symbol	Description	Definition
$\mathfrak{A}$	Set of actuators	3.1.3.1
$\mathcal{X}(\mathfrak{A})$	Set of actuators variables, also used to denote the computations representing these variables	3.1.3.1

Symbol	Description	Definition
x_i	actuator's variable	3.1.3.1
$\mathbf{x}$	value for actuator variable	3.1.3.1
$\mathcal{D}_{x_i}$	domain of actuator	3.1.3.1
$\mathcal{F}(\mathfrak{A})$	Set of actuators' energy costs	3.1.3.1
e_i	Energy cost of actuator x_i, $e_i \in \mathcal{F}(\mathfrak{A})$	3.1.3.1
$\mathfrak{S}$	Set of sensors	3.1.3.2
$\mathcal{X}(\mathfrak{S})$	Set of sensor variables	3.1.3.2
s_l	Sensor variable	3.1.3.2
$\mathbf{s}_l^p$	Value for sensor variable	3.1.3.2
$\mathfrak{R}$	Set of rules	3.1.5
$\mathcal{F}(\mathfrak{R})$	Set of rule utility functions	3.1.5
r_r	Rule utility function	3.1.5
Φ	Set of environment state	3.1.3.3
$\mathcal{X}(\Phi)$	Set of environment state variables	3.1.3.3
y_j	Environment state variable $y_j \in \mathcal{X}(\Phi)$	3.1.3.3
$\mathcal{F}(\Phi)$	Set of physical models functions	3.1.3.4
ϕ	Physical model function $\phi \in \mathcal{F}(\Phi)$	3.1.4
$\overline{\phi_j}$	Arity of physical model function ϕ_j: $\overline{\phi_j} = \lvert S_{\phi_j} \rvert$	3.1.4
$\mathcal{F}(\mathbf{\Phi})$	Set of soft constraints for physical models	3.1.5
φ	Soft constraint for a physical model: $\varphi \in \mathcal{F}(\mathbf{\Phi})$	3.1.5

Distribution

Symbol	Description	Definition
c_i	Computation	4.2.1
$\mathcal{C}$	Set of computations	4.2.1
$G_\mathcal{C}$	Computation graph, $G_\mathcal{C} = \langle \mathcal{C}, E_\mathcal{C} \rangle$	4.2.1
ν	Distribution	4.2.1
$\nu(c_i)$	Agent hosting computation c_i: $a_a =$	4.2.1
$\nu^{-1}(a_m)$	Set of computation hosted on agent a_m	4.2.1
$\mathbf{com}(c_i, c_j)$	Communication load between computations c_i and c_j	4.4
$\mathcal{C}_\mathcal{X}$	Computations bound to a variable in a factor graph	4.4.2
$\mathcal{C}_\mathcal{F}$	Computations bound to a factor in a factor graph	4.4.2
$\mathbf{mem}(c_i)$	Memory footprint of computation c_i	4.4
$\mathbf{w_{max}}(a_m)$	Memory capacity of agent a_m	4.4
$\mathbf{N}(c_i)$	Set of neighbors of c_i in the computation graph.	4.23
$\mathbf{route}(m, n)$	Communication cost between agents a_m and a_n	4.23

Symbol	Description	Definition
$\mathbf{com_a}(c_i, c_j, a_m, a_n)$	Communication cost between the computation c_i hosted on agent a_m and c_j on a_n	4.23
$\mathbf{cost}(a_m, c_i)$	Hosting cost when placing computation c_i on agent a_m	4.5.2

Replication & Repair

Symbol	Description	Definition
$\rho(c_i)$	Agents hosting a replica of c_i	5.5.1
$\mathcal{C}_c$	Set of candidate computations that could be migrated	5.5.4
$\mathcal{C}_c^m$	Set of candidate computations that could be migrated to agent a_m	5.5.4
$\mathcal{A}_c$	Set of all candidate agents, that could host a computation in $\mathcal{C}_c$	5.5.4
$\mathcal{A}_c^i$	Set of candidate agents, that could host c_i	5.5.4
$\mathcal{A}[a_k]$	Neighborhood for a_k	5.4.1
$E[a_k]$	Neighborhood edges for a_k	5.4.1
$\mathcal{C}[a_k]$	Neighborhood computations for a_k	5.4.1

B Glossary

6LowPan IPv6 over Low-Power Wireless Personal Area Networks. 98, 148

A-DCOP Asymmetric DCOP. 23

A-DPOP Approximative DPOP. 95

A-DSA Asynchronous Distributed Stochastic Algorithm. 95, 99, 133–138, 143, 185

A-MaxSum Asynchronous MaxSum Algorithm. 30, 47, 49, 50, 53, 55, 94, 95, 99, 133–139, 144, 158, 185

AAMAS International Conference on Autonomous Agents and Multiagent Systems. 3, 157

ABT Asynchronous Backtracking. 98

ADOPT Asynchronous Distributed OPTimization. 26, 98, 143

AFB Asynchronous Forward Bounding. 26

AI Artificial Intelligence. 6, 10, 13

AMAS Adaptative Multi-Agent Systems. 13, 40

AmI Ambient Intelligence. 5–10, 13, 14, 32–38, 42, 43, 50, 52, 67, 75, 76, 79, 87, 89, 94, 139, 153, 155, 156, 158, 159, 183

AnyPop Anytime DPOP. 94, 95

BDI Belief, Desire and Intention. 13, 14, 142

BSP Bulk Synchronous parallel. 76

CGDP Computation Graph Distribution Problem. 78, 79, 145

CGDP-NDP Newcomer Decision Problem for the Computation Graph Distribution Problem. 116, 120, 122, 124, 125

CoAP Constrained Application Protocol. 98, 148

ConcFB Concurrent Forward Bounding. 26

COP Constraint Optimization Problem. 15–18

CoRE Constrained RESTful Environments. 98

CRP Constraint Reasoning Problem. 15–17, 19

CSP Constraint Satisfaction Problem. 15–18, 32, 33

DAI Distributed Artificial intelligence. 10, 13

DBA Distributed Breakout Algorithm. 29, 143

DCOP Distributed Constraint Optimization Problem. 2, 3, 14, 18–20, 22–28, 30–35, 42–44, 46–55, 57, 58, 61, 63–67, 71, 72, 75–77, 79, 81, 82, 85, 87, 89–96, 98, 99, 102, 111–115, 121, 126, 127, 130, 131, 133, 137, 139, 141–143, 145, 148–153, 155, 156, 158, 159, 183, 184

DCP Dynamic Constraint Reasoning. 34, 91, 143

DCSP Distributed Constraint Satisfaction Problem. 17–19, 22, 44

Dec-MDP Decentralized Markov Decision Process. 12

Dec-POMDP Decentralized Partially Observable Markov Decision Process. 12

DFS Tree Depth-First Search Tree. 19, 21, 31, 53, 183

DMCM DCOP Model for Computation Migration. 112–115, 121, 130, 131, 133, 139, 145, 154, 156, 158, 185

DPOP Distributed Pseudo-tree Optimization Procedure. 28, 30, 47–53, 55, 94, 95, 99, 143

DRPM Distributed Replica Placement Method. 104, 106, 107, 110, 111, 113–115, 121, 126–130, 133, 134, 136–139, 145, 154, 156, 158, 184, 185

DSA Distributed Stochastic Algorithm. 28, 29, 47, 49, 50, 53, 55, 80, 94, 95, 99, 127, 128, 130–133, 143, 151, 152, 185

Dyn-DCOP Dynamic DCOP. 23, 91–95, 139

DynCSP Dynamic Constraints Satisfaction Problem. 32

EASSS European Agent Systems Summer School. 3, 157

FG Factor Graph. 17

GDBA Generalized DBA. 143

GDL Generalized Distributive Law. 26

GH-CGDP Greedy Heuristic for CGDP. 80, 82, 83, 85–87, 129, 130, 137, 151

GH-SECP-CGDP Greedy Heuristic for SECP Constraint Graph Distribution. 47, 68, 69, 80, 87, 145

GH-SECP-FGDP Greedy Heuristic for SECP Factor Graph Distribution. 47, 69, 80, 122–125, 145

IJCAI International Joint Conference on Artificial Intelligence. 2, 156

ILP Integer Linear Program. 69–75, 78–82, 84, 99, 155, 184

ILP-CGDP Integer Linear Program for CGDP. 79, 80, 82, 83, 85, 87, 97, 99, 101, 102, 111, 115, 116, 121, 145

ILP-CGDP$[a_k]^+$ Integer Linear program for CGDP restricted to the neighborhood. 115, 116, 121–125

ILP-CGDP$[a_k]^-$ Integer Linear program for CGDP restricted to the neighborhood. 101, 121, 122, 124, 125

ILP-SECP-CGDP Integer Linear Program for SECP Constraint Graph Distribution. 71, 80, 81, 87, 122, 145

ILP-SECP-FGDP Integer Linear Program for SECP Factor Graph Distribution. 73, 80, 81, 122, 123, 145

IoT Internet of Things. 3, 6–8, 33, 34, 75–79, 81, 82, 87, 89, 94, 126, 127, 130, 139, 143, 153, 155–159

IQP-CGDP-NDP Integer Quadratic Problem for CGDP-NDP. 120

JADE Java Agent Development Environment. 13

JFSMA Journées Francophones sur les Systèmes Multi-Agents. 2, 3, 154, 156, 157

MAC Multi-Agent Coordination. 33

MAS Multi-Agent Systems. 10–14, 17, 18, 33, 34, 38, 42, 58, 141–143, 145, 148

MaxSum MaxSum Algorithm. 28–30, 47, 49, 55, 64, 80, 95, 122, 128, 129, 144

MGM Maximum Gain Message. 28, 29, 47, 49, 50, 53, 55, 99, 115, 133, 143

MGM-2 Maximum Gain Message with 2-coordination. 47, 49–53, 55, 115, 130–134, 136–139, 143, 158, 183, 185

MO-DCOP Multi-objective DCOP. 23, 42

MPD Markov Decision Process. 12

NCBB No Commitment Branch and Bound. 144

NEXP Non-Deterministic Exponential. 12

NFV Network Function Virtualization. 76, 158

OptAPO Optimal Asynchronous Partial Overlay. 27

OptMAS Optimization in Multiagent Systems. 3, 156, 157

OSGi Open Service Gateway Intiative. 8

P-DCOP Probabilistic DCOP. 23

PC-DPOP Partial Centralization DPOP. 27

PFIA Plate-Forme Intelligence Artificielle. 3, 157

Q-DCOP Quantified DCOP. 23

QKP Quadratic Knapsack Problem. 120, 121

QMKP Quadratic Multiple Knapsack Problem. 104

RS-DPOP Reviewed Super-stabilizing DPOP. 94, 95

S-DPOP Self-stabilizing DPOP. 94

SECP Smart Environment Configuration Problem. 2, 35, 41–46, 48–55, 61, 67–81, 85–87, 89–97, 100–102, 121, 122, 124–126, 128–130, 137–139, 150, 155, 156, 158, 183–185

SHE Smart Home Environment. 6, 7, 13, 14, 32–34, 38, 67, 98

SOA Service Oriented Architecture. 13, 14, 32

SPOF Single Point Of Failure. 8

SyncBB ynchronous Branch and Bound. 144

UCT Upper Confidence bound for Trees. 27

VNF Virtual Network Function. 76

VNF-FG VNF Forwarding Graph. 76

List of Figures

École Nationale Supérieure des Mines
de Saint-Étienne

NNT: 2019LYSEM023

Pierre RUST

Autonomous and Spontaneous Coordination between Smart Connected Objects

Speciality: Computer Science

Domain: Artificial Intelligence

Keywords: Multi-Agent Systems, DCOP, autonomy, coordination, distribution, optimization, self-repair, self-adaptation, resilience,

Abstract:

Smart Home, Ambient Intelligence and the Internet-of-Things involve a large number of connected objects, with heterogeneous computing and communication capabilities. The high-level functionalities offered by these systems are based on the services rendered by several of these objects in a joint manner; the coordination of their actions is therefore essential. In the current systems, this coordination is implemented via a centralized entity, the connected objects are then only used as simple effectors or sensors.

This thesis examines cooperation and coordination mechanisms, in a decentralized and autonomous way, between these objects. Based on a Multi-Agent System approach called Distributed Constraints Optimization (DCOP), these objects coordinate their actions to achieve one or more objectives corresponding to the user's requirements. In this context, we underline the importance of distributing the decisions to be taken by these various agents and we present several methods for choosing a satisfactory distribution against the characteristics of the targeted systems.

Finally, since these systems are highly dynamic by nature, we present several solutions to manage the changes that may occur, both in terms of the environment and the agents themselves. In particular, we are committed to making these systems resilient, so that they can continue to operate even in the event of the disappearance of several agents. Several autonomous system repair mechanisms, based on distributed decision replication and decision making, are presented and evaluated.

École Nationale Supérieure des Mines
de Saint-Étienne

NNT : 2019LYSEM023

Pierre RUST

Coordination spontanée et autonome entre objets intelligents connectés

Spécialité de doctorat : Informatique

Discipline : Intelligence Artificielle

Mots clefs : Systèmes Multi-agents, DCOP, autonomie, coordination, distribution, optimisation, auto-réparation, résilience

Résumé :

La Smart Home, l'Intelligence Ambiante et l'Internet des Objets impliquent un grand nombre d'objets connectés, dotés de capacités hétérogènes de calcul et de communication. Les fonctionnalités de haut niveau offertes par ces systèmes s'appuient sur les services rendus par plusieurs de ces objets de manières conjointe ; la coordination de leurs actions est donc indispensable. Dans les systèmes actuels, cette coordination est mise en œuvre via une entité centralisée, les objets connectés n'étant alors utilisés que comme de simples effecteurs ou capteurs.

Cette thèse étudie les mécanismes de coopération et de coordination, de manière décentralisée et autonome, entre ces objets. En s'appuyant sur une approche issue des Systèmes Multi-Agents, l'optimisation distribuée sous contraintes (DCOP), ces objets coordonnent leurs actions pour atteindre un ou plusieurs objectifs correspondant aux souhaits de l'utilisateur. Dans ce contexte, nous soulignons l'importance de la distribution des décisions à effectuer par ces différents agents et présentons plusieurs méthodes permettant de choisir une distribution satisfaisante en regard des caractéristiques des systèmes ciblés.

Finalement, ces systèmes étant par nature hautement dynamiques, nous présentons plusieurs solutions pour gérer les changements pouvant survenir, tant au niveau de l'environnement que des agents eux-mêmes. En particulier, nous nous attachons à rendre ces systèmes résilients, afin qu'ils puissent continuer à opérer même dans le cas de la disparition de plusieurs agents. Plusieurs mécanismes de réparation autonome du système, basés sur la réplication des décisions et la prise de décision distribuée, sont présentés et évalués.