UNIVERSITY OF SOUTHAMPTON
Faculty of Physical and Applied Sciences
Electronics and Computer Science
Agents, Interaction, and Complexity Group

Multi-Agent Coordination for Dynamic Decentralised Task Allocation

by Kathryn S. Macarthur

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

Supervisors: Prof. N. R. Jennings and Dr. S. D. Ramchurn
Examiner: Dr J. A. Rodrìguez Aguilar and Dr. A. C. Rogers

December 2011

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Kathryn S. Macarthur

Coordination of multiple agents for dynamic task allocation is an important and challenging problem, which involves deciding how to assign a set of agents to a set of tasks, both of which may change over time (i.e., it is a dynamic environment). Moreover, it is often necessary for heterogeneous agents to form teams to complete certain tasks in the environment. In these teams, agents can often complete tasks more efficiently or accurately, as a result of their synergistic abilities.

In this thesis we view these dynamic task allocation problems as a multi-agent system and investigate coordination techniques for such systems. In more detail, we focus specially on the distributed constraint optimisation problem (DCOP) formalism as our coordination technique. Now, a DCOP consists of agents, variables and functions — agents must work together to find the optimal configuration of variable values. Given its ubiquity, a number of decentralised algorithms for solving such problems exist, including DPOP, ADOPT, and the GDL family of algorithms. In this thesis, we examine the anatomy of the above-mentioned DCOP algorithms and highlight their shortcomings with regard to their application to dynamic task allocation scenarios. We then explain why the max-sum algorithm (a member of the GDL family) is the most appropriate for our setting, and define specific requirements for performing multi-agent coordination in a dynamic task allocation scenario: namely, scalability, robustness, efficiency in communication, adaptiveness, solution quality, and boundedness.

In particular, we present three dynamic task allocation algorithms: fast-max-sum, branch-and-bound fast-max-sum and bounded fast-max-sum, which build on the basic max-sum algorithm. The former introduces storage and decision rules at each agent to reduce overheads incurred by re-running the algorithm every time the environment changes. However, the overall computational complexity of fast-max-sum is exponential in the number of agents that could complete a task in the environment. Hence, in branch-and-bound fast-max-sum, we give fast-max-sum significant new capabilities: namely, an online pruning procedure that simplifies the problem, and a branch-and-bound tech-

nique that reduces the search space. This allows us to scale to problems with hundreds of tasks and agents, at the expense of additional storage. Despite this, fast-max-sum is only proven to converge to an optimal solution on instances where the underlying graph contains no cycles. In contrast, bounded fast-max-sum builds on techniques found in bounded max-sum, another extension of max-sum, to find bounded approximate solutions on arbitrary graphs. Given such a graph, bounded fast-max-sum will run our iGHS algorithm, which computes a maximum spanning tree on subsections of a graph, in order to reduce overheads when there is a change in the environment. Bounded fast-max-sum will then run fast-max-sum on this maximum spanning tree in order to find a solution. We have found that fast-max-sum reduces the size of messages communicated and the amount of computation by up to 99% compared with the original max-sum. We also found that, even in large environments, branch-and-bound fast-max-sum finds a solution using 99% less computation and up to 58% fewer messages than fast-max-sum. Finally, we found bounded fast-max-sum reduces the communication and computation cost of bounded max-sum by up to 99%, while obtaining 60–88% of the optimal utility, at the expense of needing additional communication than using fast-max-sum alone. Thus, fast-max-sum or branch-and-bound fast-max-sum should be used where communication is expensive and provable solution quality is not necessary, and bounded fast-max-sum where communication is less expensive, and provable solution quality is required.

Now, in order to achieve such improvements over max-sum, fast-max-sum exploits a particularly expressive model of the environment by modelling tasks in the environment as function nodes in a factor graph, which need to have some communication and computation performed for them. An equivalent problem to this can be found in operations research, and is known as scheduling jobs on unrelated parallel machines (also known as $R||C_{\max}$). In this thesis, we draw parallels between unrelated parallel machine scheduling and the computation distribution problem, and, in so doing, we present the spanning tree decentralised task distribution algorithm (ST-DTDA), the first decentralised solution to $R||C_{\max}$. Empirical evaluation of a number of heuristics for ST-DTDA shows solution quality achieved is up to 90% of the optimal on sparse graphs, in the best case, whilst worst-case quality bounds can be estimated within 5% of the solution found, in the best case.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

$\chi_i(f_j)$ The application-specific time required for agent a_i to compute and communicate for function f_j, page 60

$\delta_j(a_i, C_j)$ The contribution of agent a_i to coalition C_j when performing task t_j, page 57

Δ_k The domain of variable y_k, page 62

$\mathbf{Y}_i$ A configuration of the variables in Y_i, page 63

$\mathcal{A}$ The set of agents, page 56

$\mathcal{A}_j$ The set of agents that can perform task t_j, page 57

$\mathcal{D}$ The set of factor graph variable domains, page 57

$\mathcal{E}$ The set of edges in a factor graph, page 59

$\mathcal{F}$ The set of functions, page 58

$\mathcal{N}(a_i)$ The set of agents with which a_i shares at least one variable, page 63

$\mathcal{S}$ The set of all coalition structures, page 57

$\mathcal{S}^*$ The coalition structure which maximises the global utility, page 57

$\mathcal{T}$ The set of tasks, page 56

$\mathcal{T}_i$ The set of tasks agent a_i can perform, page 57

$\mathcal{X}$ The set of factor graph variables, page 57

$\mathcal{Y}$ The set of all junction graph variables, page 62

ψ_i Agent a_i's potential function, page 62

$\zeta_i(Y_i)$ The marginal function of agent a_i, page 63

A_j The set of agents that can perform the computation of function f_j, page 60

C A coalition of agents, page 57

C_j The coalition performing task t_j, page 57

F_i The subset of $\mathcal{F}$ that a_i can perform the computation of, page 60

M The set of all possible mappings, page 61

m A mapping from functions to be computed to agents to compute them, page 61

$m(a_i)$ The set of functions assigned to a_i under mapping m, page 61

$V(C_j, t_j)$ The utility gained by coalition C_j performing task t_j, page 57

Y_i The set of variables in a_i's clique, page 62

Acronyms

ABT Asynchronous backtracking.

ADOPT Asynchronous distributed constraint optimization.

BFMS Bounded fast-max-sum.

BMS Bounded max-sum.

BnB FMS Branch-and-bound fast-max-sum.

BnB MS Branch-and-bound max-sum.

CEMTDP Conjoined, extended Markov team decision process.

CSP Constraint satisfaction problem.

DaCSA Divide and coordinate subgradient algorithm.

DALO Distributed asynchronous local optimization.

DCOP Distributed constraint optimisation problem.

DecMDP Decentralised Markov decision process.

DFS Depth-first search.

DPOP Distributed pseudotree optimisation procedure.

DSA Distributed stochastic algorithm.

DTDA Decentralised task distribution algorithm.

FDNY Fire department of New York.

FMS Fast-max-sum.

GDL Generalised distributive law.

GHS GHS.

GSB Gold, silver, bronze.

ICS Incident command system.

iGHS Iterative GHS.

MDP Markov decision process.

MMDP Multi-agent Markov decision process.

MST Minimum/maximum spanning tree.

MTDP Markov team decision process.

NYPD New York police department.

ODP Online domain pruning.

OPGA Overlapping potential game approximation.

POMDP Partially observable Markov decision process.

SDPOP Superstabilizing distributed pseudotree optimisation procedure.

ST Spanning tree.

ST-DTDA Spanning tree decentralised task distribution algorithm.

WTC World trade center.

Declaration of Authorship

I, Kathryn Sarah Macarthur,
declare that the thesis entitled
Multi-Agent Coordination for Dynamic Decentralised Task Allocation
and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published in a number of conference and journal papers (see Section 1.4 for a list).

Signed:
Date:

Acknowledgements

I thank my supervisors, Nick and Gopal, for their expert help and guidance throughout my PhD. Also, thanks to the IAM/AIC group at the University of Southampton for giving me the opportunity to do the PhD, and to BAE Systems and the EPSRC for funding it via the Aladdin project.

I thank my coauthors, Alessandro, Meritxell and Ruben, for their invaluable technical guidance, support, and friendship, and my (former) colleagues who have encouraged and guided me through the last three years: particularly, Archie, Colin, Francesco, Harry, Heather, Long, Seb, Simon, and Talal.

I thank Mr Maggs, for believing in me and encouraging me to aim higher, all those years ago at school, and to all the lovely ECS academics and lecturers for their encouragement through my undergrad, and, in so doing, making me want to stay on for this PhD.

I thank my friends, for putting up with me and always being able to make me laugh. I especially want to thank Adam, Ali, Ben, Cally, Jen, Kate, Lau, Rachie and Reena for putting up with the vast majority of my rants, for keeping me sane, and for sticking by me through thick and thin. Also, thank you Caroline and Mr Dyson for fellow PhD student rants that no-one else could understand!

I owe a great debt of gratitude to the best partner I could ever wish for, Marcus: I literally could not have done this PhD without you. Your unwavering support throughout the past five years has enabled me to grow stronger as a person, and to achieve things I never thought possible.

Last, but by no means least, I thank my mum, my dad, my sister Helen, and my brother Rich: my wonderful, loving family, without whom I would never have been able to do this. Words cannot express how much I love and appreciate each of you, how proud I am of each of you, and how much you have helped me to achieve this great milestone. I could not have wished for a better start in life, or for a better family to share it with. You made me what I am today, and I did this for you.

I sincerely thank you all from the bottom of my heart.

"There are at least three machines available on this computer.
Do not get them mixed.

1. *The word processor which combines typewriter and filing cabinet*
2. *The internet which admits you to sources of information which are vast and come in different forms*
3. *The E mail which allows you to communicate in writing with any one whose E mail number you know or can find out. There is no E Mail directory.*

Some of these functions overlap and provide the same facilities, but you will end in confusion if you forget which source you are using."

For Grandpa, Revd. Arthur Leitch Macarthur MA MLitt OBE,
and his good friend Jeeves.

Chapter 1

Introduction

Dynamic task allocation is an important and challenging problem, which involves deciding how to assign a set of actors to a set of tasks, both of which may change over time (i.e., it is a dynamic environment). One example of such an environment is that following a disaster, such as an earthquake or hurricane, the rescuers (actors) must be assigned civilians to rescue (tasks) in order to save as many civilians as possible, whilst reacting quickly to changes in the environment. These changes come from events such as the discovery of more civilians to rescue, new rescuers appearing, or existing rescuers becoming incapacitated. Another such example can be found in target tracking, where a number of sensors (agents) must be assigned to targets (tasks) such that sensor coverage is maximised. The dynamism in this environment comes from new targets being discovered, existing targets leaving the tracking area, new sensors becoming available, and the failure of existing sensors.

This dynamism is inherent in many real-world task allocation environments, and so, in order to apply mechanisms to find solutions to task allocation problems in the real world, such mechanisms must be able to adapt to changes in the environment whilst operating in order to update their solutions. In addition to this, the likelihood of the failure of actors in these environments means that assignments cannot simply be computed by a single actor, since if that actor were to fail, then the entire assignment would have to be re-calculated by another actor. In order to avoid such a central point of failure, it is important for solution computation to be *decentralised*: i.e., spread among all of the agents in the environment.

Against this background, in this work, we present a decentralised multi-agent systems approach to dynamic task allocation. To this end, in this chapter, we first outline the dynamic task allocation problem, and identify key requirements of a decentralised solution approach (Section 1.1). Next, in Section 1.2, we describe the multi-agent systems paradigm: what it is, how the agents coordinate, and how it can be used to model real-world dynamic task allocation domains such as the disaster response example given

earlier. Given this background, we identify our specific aims and objectives in Section 1.3, and explain the contributions we have made to the field in Section 1.4. We then present an outline of this thesis in Section 1.5.

1.1 Decentralised Task Allocation in Dynamic Environments

Task allocation problems consist of a set of actors and a set of tasks. The actors will gain some utility from being assigned to a task — thus, an optimal solution to a task allocation problem is an allocation assigning each actor to a task such that the total utility gained by all actors is maximised (i.e., we wish to maximise global utility). It is generally likely that not all actors will be able to perform all tasks due to physical limitations such as distance between actors and tasks, or operational limitations such as the inability of an actor to do a particular task. Thus, it is likely that task allocation problems will exhibit *sparsity* in task-actor interactions. This can be seen in the disaster response example given earlier — it is unlikely that every rescuer will feasibly be able to reach every civilian in need of rescuing due to routes being blocked off by rubble or fires, and so, will instead choose from a limited subset of all civilians to be rescued. Similarly, with target tracking, it is unlikely that a fixed sensor will be capable of tracking every target in an environment, and instead will choose a target from a restricted set of available targets.

An interesting set of domains for task allocation problems are *dynamic* environments, where the sets of tasks and actors, and the links between them, can, and will, change over time. These *dynamic task allocation problems* are extremely prevalent in real-world scenarios — for example, dynamic task allocation problems can be found in the allocation of sensing tasks to mobile sensing robots, where the dynamism comes from the movement of the robots and changing tasks (Zheng and Koenig, 2008), and in the assignment of rescue tasks to ambulances, where the dynamism comes from new rescue tasks appearing, the ambulances moving around, and from tasks being completed (Shehory and Kraus, 1998). In these problems, and many others besides, coordination amongst the actors is especially important because it enables a group of heterogeneous actors to find the best possible solutions as the environment evolves. In using the term *coordination*, we refer to the resolution of interdependencies between activities or different organisational units (Smith and Dowell, 2000). For example, in the case of disaster management, rescuers need to coordinate with one another in order to maximise the number of civilians they rescue (where the civilians to be rescued are the interdependencies), thus avoiding scenarios where all rescuers go to the same civilian, abandoning all others. Again, this can be found in target tracking — if all sensors were to track the same target (targets tracked by sensors are the interdependencies here), then this is likely to be less useful than covering a larger number of targets. There are two key ways of performing this

coordination: the first of which is known as a *decentralised* manner of finding a solution, which refers to actors communicating amongst themselves to compute an assignment. The alternative to this is a *centralised* approach, where some central entity computes an assignment and communicates it to all actors — however, such a centralised approach creates a central point of failure, which is not ideal in the dynamic environments we consider, where failure of actors is likely.

Given all this, it is important to study the key characteristics of such dynamic task allocation scenarios, so that we can identify the requirements for our own decentralised dynamic task allocation approach. We do this using search and rescue in disaster response as a running example, since it is has become such an important area of research in recent years, illustrated by events such as the attacks on New York City in 2001, which killed over 2600 people (9/11 Commission, 2004), the effects of Hurricane Katrina on New Orleans in 2005, which resulted in over $80 billion of damage (Knabb et al., 2006), and the 2011 Tōhoku earthquake in Japan, which resulted in the deaths of over 15 thousand people, and the collapse of over 100 thousand buildings (National Police Agency of Japan, 2011). As these episodes illustrate, after man-made and natural disasters occur, the environment around them evolves quickly, so any search and rescue response effort must be efficiently managed to ensure aid arrives at a disaster area promptly. In the context of task allocation, in these scenarios the rescuers would be the actors, and civilians to rescue would be the tasks. Nevertheless, it is important to remember that these key aspects are present in many other scenarios, such as pursuit evasion (Parsons, 1976), static and mobile sensor networks (Stranders et al., 2009), and radar scheduling (Kim et al., 2010).

The first key characteristic of dynamic task allocation problems is that they often take place in environments which contain hundreds of tasks and actors. An example of the scale that dynamic task allocation problems can grow to can be found in disaster response, which is performed by hundreds of individual people from multiple different rescue agencies (such as emergency services, charities, volunteers, and private companies), each with its own objectives, properties and constraints. For example, after the 9/11 attacks on the world trade center (WTC) towers, it was necessary for hundreds of fire department of New York (FDNY) ambulances, private (hospital-owned) ambulances, New York police department (NYPD) officers, and volunteers from the public to work together in order to save as many lives as possible (Simon and Teperman, 2001). An example of coordinating large-scale rescue efforts can be found when comparing the disaster response mechanism in place in the UK (London Emergency Services Liason Panel, 2007), which is known as the gold, silver, bronze (GSB) model, and that of the USA (US Department of Homeland Security, 2004), Australia (Australasian Fire Authority Council, 2004), New Zealand (Ministry of Civil Defence and Emergency Management, 2007), and British Columbia, Canada (Provincial Emergency Program, 2007), which is known as the incident command system (ICS). The GSB model is strictly hierarchical,

having three levels of command: operational (bronze), tactical (silver) and strategic (gold). Decisions are made by those at the top level of this model (gold), who may only communicate with those on the ground through an intermediary level of command. This means that the system has the potential to be slower and less adaptive than may be preferable in such situations. Despite this, the strong hierarchical nature of the GSB model does ensure that all responding agencies are immediately certain of their responsibilities. The ICS model, which is used in a larger number of countries, focuses on three modules: planning, operations, and logistics. In more detail, the planning module is responsible for collecting, evaluating, and disseminating tactical information related to the incident, and for preparing and documenting Incident Action Plans (IAPs). Next, the operations module is responsible for all operations directly applicable to the primary mission of the response to a disaster. Finally, the logistics module is responsible for providing facilities, services and materials for the response to a disaster. Given all this, GSB and ICS are practical examples of centralised and decentralised task allocation solution mechanisms. In more detail, the reliance on hierarchy in GSB means that if those at the top of the hierarchy (gold) were to be incapacitated, or if communication lines to their office were to be cut, then the mechanism would need to allocate a new gold level of command, and begin again. Thus, GSB has a single point of failure, and, as such, is centralised. In contrast to this, ICS is decentralised since communication and coordination takes place amongst the agencies involved, and does not rely on any central point of failure. In addition to this, ICS facilitates the exploitation of sparsity in interactions, which is important, since, as it is unlikely that all actors will be able to do all tasks, this allows decisions to be made more quickly. Given all this, in this work, we aim to achieve something similar to the ICS model, because it is decentralised and produces decisions quickly, even in large-scale environments.

Next, since we consider dynamism in terms of both actors and tasks, it is important that we consider actor failure: both in terms of operational failure (i.e., an actor becoming unable to perform any more tasks because of injury, death or being otherwise incapacitated), and in terms of communication failure (for example, an actor's communication device could be faulty, communication antennae could go down, or there could be interference on communication lines). Therefore, since it is likely that such failures will happen, approaches with a single point of failure, are not appropriate for use in such environments, since their failure will bring the whole system to a standstill. Returning to our disaster response example, it is likely that the operation of agencies in such perilous environments will lead to agency units failing due to destruction by the environment. For example, rescue vehicles could be crushed by falling debris, or rescuers could perish in fire following an earthquake. Thus, in such scenarios, it is important that actors becoming unable to continue their computation do not have a significant negative impact on the overall solution computation.

The next characteristic to consider is communication amongst actors — specifically,

that available communication channels may be unreliable and/or expensive to use. For example, in disaster response scenarios, many actors may need to communicate in order to ascertain a structure of command and set of responsibilities, whilst sharing tasks and resources. Hence, unreliable and expensive communication can make coordination amongst agencies even more difficult — for example, emergency communication infrastructure was taken out with one of the WTC towers, making communication amongst agencies difficult and unreliable (Simon and Teperman, 2001). Thus, solution mechanisms for dynamic task allocation problems must be communication-efficient, and avoid superfluous communication.

Perhaps the most important characteristic of dynamic task allocation environments is their dynamism. For example, after man-made and natural disasters occur, the environment around them evolves quickly, and in unpredictable ways, so any response effort must be efficiently managed to ensure aid arrives at a disaster area promptly and be capable of dynamically re-allocating resources at short notice. For example, building collapse will occur over time following an earthquake, as a result of the spread of fires, and even other building collapse, and such building collapse could result in the incapacitation or death of civilians to be rescued, or the rescuers themselves. In addition to this, new civilians to be rescued could be discovered by workers, and new rescuers could arrive at the scene to help. Now, these changes can be predicted to some extent (for example, by using established results on fire spread in specific types of building, or through communication with arriving agencies), but it is unlikely to be predicted with complete accuracy. This means that changes are likely to be sudden, thus exemplifying the need for the capability to re-allocate resources at short notice. Another example of this is the uncertainty that follows a terrorist attack — none of the rescue agencies on the ground will have any idea whether or not further attacks will occur within a short time, or how the situation will evolve. One simple solution to cope with changes in the environment is to recompute solutions from scratch at the arrival or departure of an actor or task; however, this is incompatible with the aforementioned problem of communication unreliability and cost. Therefore, there is a need for a solution mechanism to recover gracefully from changes in the environment as and when they occur, whilst avoiding extraneous communication and computation.

In sum, we can identify a number of common characteristics of dynamic decentralised task allocation problems, which must be addressed by solution mechanisms. Continuing to use disaster management as an example, these characteristics are as follows:

Coordination of multiple heterogeneous actors — A large number of actors and tasks, each with their own capabilities and requirements, are likely to be present in a real-world dynamic task allocation problem. For example, in a disaster, a large number of agencies may rescue civilians, each with multiple units under its control. Hence, coordination must occur amongst these agencies in order for them to maximise the number of lives saved. Broadly, this coordination can be done

through a hierarchical, centralised system such as GSB or it can be done in a decentralised manner as in ICS.

Unit failure — Dynamic task allocation solution mechanisms must be robust to actor failure, and so must not rely on a centralised entity to perform solution computation, since if the centralised entity were to fail then the solution mechanism would have to start again from scratch elsewhere. Thus, such computation must be distributed amongst actors. This is particularly important in dangerous scenarios such as disaster management, where it is likely that some or all of the available rescue units or equipment could break down. Thus, the solution mechanism employed by the actors must be robust to such failures.

Bandwidth limitations — It is important for solution mechanisms to be efficient in communication, in a situation where many different issues could lead to areas of restricted, noisy or delayed communication. This was particularly evident in 9/11 (Simon and Teperman, 2001), where key communication infrastructure was destroyed.

Rapidly changing environment — In a dynamic task allocation environment which is always changing, the ability to adapt quickly is important. For example, in disaster management scenarios, when units are unaware of the conditions in areas of the environment until they enter them, the ability to immediately change allocations is key.

Now, having outlined these characteristics, in the next subsection we elaborate on the methodology that we will use to address them.

1.2 Multi-Agent Coordination

The multi-agent systems paradigm revolves around the decentralised coordination of multiple *intelligent agents*, which are equivalent to the actors in our dynamic task allocation problems. In more detail, an intelligent agent, as defined by Wooldridge and Jennings (1995), is autonomous, reactive, proactive, and has some degree of social ability. Here, autonomy is defined as the ability to complete a task without external control or influence, reactivity as perception of its environment and response to that perception, pro-activity as taking the initiative to advance towards goals set, and social ability as the capacity to communicate and interact with other agents. Of these properties, autonomy is generally considered the most prominent, given the general understanding of the word 'agent' as being that of an entity which acts with no outside intervention. This general definition of an autonomous agent fits well in our example domain, where a rescuer in a disaster could be equipped with a smartphone containing agent software, which could

coordinate with other smartphone agents and help guide the rescuer's decision as to what to do next.

Given this, multi-agent systems are systems consisting of multiple intelligent agents working toward individual, and possibly joint, objectives. In the multi-agent systems methodology, a systems approach to modelling is taken, working on the assumption that taking a view of the system as a whole proves more beneficial than viewing each agent individually. This approach, therefore, enables the engineering of desirable system-level emergent properties. Moreover, the inherent decentralisation of multi-agent systems may also be used to spread computational load across the entire system, thus avoiding a single point of failure.

In a multi-agent system, coordination is crucial in order to ensure that actions taken by all agents will contribute toward their individual or common goal. When using multi-agent coordination in complex and dynamic domains, such as dynamic task allocation, the ability to find high quality solutions quickly (and, preferably, provide a bound on this quality) and adapt to changes in the environment of any method chosen to achieve this coordination is paramount. Thus, we consider each of four key classes of coordination technologies: game theoretic mechanisms, auctions, Markov decision processes (MDPs), and distributed constraint optimisation problem (DCOP) formulations. They are described below (see Chapter 2 for more details):

Game theoretic mechanisms centre around the design of rules of interaction for a number of actors (Von Neumann and Morgenstern, 1944). There are two main branches of game theoretic mechanisms: cooperative and noncooperative. In both branches, the actors can take a number of actions; however, the branches differ in the actors' aims. In the former, agents form coalitions to perform actions in order to maximise their joint reward (generally referred to as transferable utility in the literature), which must then somehow be split amongst the agents: a task which can be computationally complex in itself. In contrast, in noncooperative game theory, each combination of actions taken by agents in the system leads to a certain reward for each agent, and the aim of each agent is to achieve the highest reward it can (i.e., the agents are selfish). In noncooperative game theory, this often equates to finding what is known as a Nash equilibrium: the combination of player actions that results in every player performing the best that they can against each other, which, in practice, is often intractable since it requires searching the space of action combinations, which grows exponentially. An example of noncooperative game theoretic techniques used for multi-agent coordination for dynamic task allocation can be found in the work of Chapman et al. (2009). In this work, the changing environment must be represented by a generally intractable type of game, the Markov game. This intractability is overcome by approximating a Markov game using a sequence of potential games, for which an equilibrium can

be found. However, this approximation derives agent utilities from an approximation of the global utility. This approximation means that quality guarantees are not provided on any equilibria found, as they could lie in a local maximum, as opposed to a global maximum. In order to find a global maximum, a significant amount of computation must be incurred, which prevents any algorithm from being scalable in the number of agents, or reliable in environments where agents are likely to be removed from the problem. We choose not to use cooperative or noncooperative game theoretic techniques in our work because neither is a natural fit for our dynamic task allocation problem. In more detail, the use of cooperative game theory requires the additional computational complexity of distributing transferable utility amongst agents on top of the complexity of finding a solution, and noncooperative game theory would require creating an approximation of our problem involving selfish agents (as opposed to cooperative ones). In addition to this, finding the Nash equilibrium does not necessarily equate to finding the global maximum: it can be a local maximum, and in dynamic task allocation we are concerned with finding a global maximum (or something very close to it).

Auctions and market-based control mechanisms involve buying and selling goods or services by offering them to bidders, and selling the goods or services to the highest bidder. In multi-agent systems, auctions consist of a number of items and a number of agents that wish to buy one or more items being sold by some other agents. An auction is started for each item in the market, in which agents wanting an item place bids, which represent how much they are willing to pay for that item. In general, the agent that places the highest bid wins the item, although the price they will pay for it will depend on the type of auction — for example, English, Dutch or Vickrey (Vickrey, 1961). Auction mechanisms have been used for multi-agent coordination in the past, for example, for finding the shortest distance for a team of robots to travel in static environments (Lagoudakis et al., 2004) and as a decision-making process for an agent assigning ambulances to attend to incidents (López et al., 2005), amongst others. However, there are a number of issues preventing the use of auction mechanisms for our purpose: specifically, the complexity that can be involved with determining the winner of a multi-dimensional auction, the issue of how to distribute transferable utility amongst bidder and seller agents, and the fact that auction mechanisms rely on communication between the bidders and auctioneer, which can lead to slow decision making when communication is delayed, or if the auctioneer were to fail. This is, therefore, not a natural fit for dynamic task allocation problems where agents need to co-operate and make decisions quickly.

MDP techniques, such as partially observable Markov decision processes (POMDPs) (Kaelbling et al., 1998), multi-agent Markov decision processes (MMDPs) (Boutilier, 1996), and Markov team decision processes (MTDPs) (Pynadath and Tambe, 2002) are generally used for modelling sequential decision making under various

degrees of uncertainty. Such techniques have been applied to areas useful to disaster management, such as the conjoined, extended Markov team decision process (CEMTDP), which was developed by Paruchuri et al. (2004) in order to solve resource-allocation problems involving multiple agents. MDPs model coordination explicitly, by searching all potential system states in order to find the optimal action for each agent in each possible state of the system. However, finding the optimal action is NP hard — the number of states to search can make the search intractable, or very costly when scaling up to high numbers of agents, and so, solutions are often approximated, or can be optimal but with exponential computation, which is often infeasible in real-world environments such as disaster management environments, which commonly have few available resources. For these reasons, we choose not to use MDPs.

DCOP formulations consist of multiple cooperative agents, which control a set of variables, and work together to optimise a set of constraints upon those variables. Thus, a solution to a coordination problem assigns an action to each agent in the system. DCOPs can be used for coordination by modelling each agent as a variable whose domain consists of all the actions that an agent can take, and encoding in the objective function which agents may take which actions at what time. A solution to a DCOP takes the form of an assignment of variables to values in their domains. Decentralised solution approaches for DCOPs can find optimal or near-optimal solutions in an efficient manner. We believe that DCOP solution techniques are suitable for dynamic task allocation problems, because they are potentially very scalable (distributing computation of a solution is much more efficient and robust than one agent computing the entire solution), low cost, have the ability to operate under rapidly changing environments (Petcu and Faltings, 2005b), and are capable of giving either optimal solutions (Aji and McEliece, 2000; Petcu and Faltings, 2005a; Modi et al., 2005), or good quality bounded estimations (Farinelli et al., 2009). In more detail, the specific DCOP solution techniques we look at require the use of a graphical representation of the environment, upon which they run some preprocessing, in order to pass around utility values and calculate a solution based on those values. Such a graphical representation is a good fit to represent scenarios such as ours, which exhibit sparse interactions between agents and tasks.

Having outlined the high-level rationale for a DCOP approach, we now provide more details to justify this choice. In particular, we represent the assignment of agents to tasks as a DCOP by treating each agent as a variable, and the potential tasks it can do as its domain. The DCOP representation is flexible enough that we can represent a rapidly changing environment without the need for building a completely new representation of the environment after every change, by adding elements to or removing elements from the underlying graphical representation used by all DCOP solution techniques. This

flexibility of representation means that DCOP solution mechanisms should be able to recover gracefully from changes in the environment.

Given this, we have identified a number of algorithms for finding solutions to DCOPs. In more detail, these algorithms (see Section 2.2) are decentralised, and so, robust to the failure of any single agent because they do not have a central point of failure. These techniques are also often able to scale, both in computation and communication, if the connections between variables and functions are sparse, as they can be in task allocation. For example, decentralised algorithms such as the generalised distributive law (GDL) framework (Aji and McEliece, 2000) (specifically, the max-sum algorithm, which is used to maximise *social welfare*: i.e., the total utility gained), distributed pseudotree optimisation procedure (DPOP) (Petcu and Faltings, 2005a), and asynchronous distributed constraint optimisation (ADOPT) (Modi et al., 2005) require agents only to contact their close neighbours when making decisions. Given this, we believe that decentralised methods, particularly the max-sum algorithm, are well suited to dynamic task allocation due to their robustness to agent failure and ability to scale well in sparse environments (we discuss this further in Section 2.2). However, there is a need for specific focus to be placed on reducing unnecessary communication and computation performed by the standard version of max-sum after a change in the environment in order for it to be applicable to a real world dynamic task allocation problem, such as a disaster management environment. In addition to this, some formulations used by max-sum lead to a problem of decentralised *computation distribution*, where tasks (which are incapable of performing any communication and computation) are represented by nodes requiring communication and computation — thus, it is important to ensure that agents are able to allocate this computation and communication, so that the solution mechanism can be considered completely decentralised.

Against this background, we now identify a number of key characteristics of dynamic task allocation problems, which are not addressed by the current state-of-the-art, and summarise how DCOP techniques for multi-agent systems will allow us to address them:

Coordination of multiple heterogeneous agencies — Each actor (for example, each rescuer in disaster response, or each sensor in target tracking) can be viewed as an autonomous, decision making entity, and therefore lends itself well to being represented as an agent in a multi-agent system, and subsequently, an agent in a DCOP, since an agent in a DCOP controls a variable representing an actor's assigned task.

Unit failure — Many algorithms to solve DCOPs are decentralised, and so, remove a single point of failure and spread decision-making and control over all agents in the system. Therefore, if a unit were to fail, the remaining units would be able to recover from the loss without much disruption. However, not all algorithms to solve DCOPs can be considered completely decentralised since their operation

means that removal of one agent whilst the algorithm is running will require the entire algorithm to be re-run from scratch (for example ADOPT (Modi et al., 2005) and DPOP (Petcu and Faltings, 2005a)).

Bandwidth limitations — Decentralised DCOP algorithms rely on local message passing amongst the agents to find solutions, which, if the messages are small in size, is ideal in situations where the range of bandwidth is limited or the communication lines are unreliable.

Rapidly changing environment — DCOPs are a very flexible representation of an environment, and take advantage of sparsity in an environment to find a solution quickly. As such, the underlying representation of the environment can, in some algorithms, be changed at will without having much effect on finding a solution (Petcu and Faltings, 2005b). If this flexibility is present, then a DCOP solution mechanism should be able to recover relatively easily from changes in the environment. Otherwise, in such a situation, some DCOP algorithms will need to recompute from scratch.

Having identified the DCOP formulation as suitable for use for our dynamic task allocation problem, we next discuss the aims and objectives of this research.

1.3 Research Aims and Objectives

The ultimate aim of this work is to develop a multi-agent coordination mechanism to solve dynamic task allocation problems in complex environments. Specifically, the mechanism should satisfy the following set of requirements:

1. **Scalability** — The mechanism should find a solution quickly and efficiently, even in large, complex environments, in which hundreds of agents and tasks are present, without significantly impacting solution quality (see requirement 5).

2. **Robustness** — The mechanism should operate such that an agent or task can be removed from or added to the environment during solution computation without causing complete re-computation amongst the other agents. In addition to this, the mechanism should also exhibit robustness in terms of communication, such that if a communication link is lossy or unreliable, the mechanism should continue solution computation without having to be completely re-run. Thus, the mechanism should be completely decentralised, in that decision making should be spread throughout all agents.

3. **Efficiency** — The mechanism should be efficient in terms of communication and computation. When changes occur in the environment, the mechanism should,

on average, require less communication and computation than it would require to completely re-run from scratch.

4. **Adaptiveness** — The mechanism must operate in real-time, and be able to adapt to changes in the environment (for example, addition or loss of agents and/or tasks) as and when they occur, without incurring superfluous communication and computation.

5. **Quality** — The mechanism should be able to provide good quality solutions regardless of the nature of the interdependencies between agents and tasks in the environment.

6. **Boundedness** — The mechanism should provide provable bounds on the solution quality achieved.

In the next section, we give our research contributions and explain how they meet these requirements.

1.4 Research Contributions

Against the objectives set in the previous section, we have formulated a generic dynamic task allocation scenario. It contains a number of agents, and a number of tasks for the agents to perform. More specifically, we show how to formulate this problem as a factor graph (see Kschischang et al. (2001)), a bipartite undirected graph (explained further in Section 2.2.1), in order to apply the max-sum algorithm to it (as previously justified in Section 1.2). Max-sum uses a decentralised message passing mechanism to find an optimal assignment of agents to tasks, and has been proven to converge to a solution over tree structured graphs. Max-sum is a good starting point for developing a mechanism, as it already meets the requirements of being scalable, decentralised, to some extent, robust to changes in the environment, and, in terms of bounded max-sum, bounded (See Requirements 1, 2, 4 and 6 from Section 1.3). However, despite this, the max-sum algorithm does not make any specific provisions for dynamic environments, and so, it is forced to re-compute the entire solution each time a change occurs in the environment, incurring unnecessary computation and communication, and so, violating requirements 3 and, to some extent, 4. This is a particularly big issue in large environments, where even the smallest of changes could trigger every agent in the environment having to communicate and compute a new solution, when none of the agents' assignments would change. For example, in the disaster response scenario discussed earlier, a new rescuer arriving at the scene should not cause all existing rescuers to stop what they are doing and decide where each should go all over again. This can be avoided, for example, by only re-computing the solution over the portion of graph effected by the change.

However, whilst max-sum is a good starting point for our work, certain factor graph formulations (where function nodes are created for each task in the environment) give rise to the problem of deciding which agents should perform computation and communication relating to each task in the environment. In more detail, since tasks themselves are abstract entities (for example, a fire to be extinguished, a civilian to be rescued, or a target to be tracked), they are unable to perform the communication and computation required for the function nodes that represent them, unlike agents (e.g., firefighters, rescuers, or sensors), which are able to perform communication and computation of the variable nodes that represent them. Thus, if multiple agents are able to perform a task, then one of the agents needs to be nominated to communicate with the other agents and perform computation for the function node representing the task. This is not a trivial decision since agents could have different computational and communicational abilities which could make certain agents compute and communicate for certain function nodes more slowly than others, which, in turn, would slow down the operation of max-sum. Thus, in order for our algorithm to be truly robust to agent failure (meeting requirement 2), and so not reliant on some central entity deciding which agents should compute for which functions, we need to solve this problem of decentralised *computation distribution* in order for max-sum to be applied in dynamic task allocation environments such as ours. Finding a decentralised solution to such a problem is an important area of research, especially since the computation distribution problem is paralleled by one known in operations research as *scheduling on unrelated parallel machines*, or $R||C_{\max}$ — for which there are currently *no* decentralised solutions.

More specifically, the contributions of this thesis are as follows:

Fast-max-sum (FMS) (Section 4.2) — the first max-sum extension to specifically reduce communication and computation incurred by re-computation in an environment which changes over time (specifically addressing Requirements 3 and 4). More specifically, we have extended max-sum to enable each agent to store its previous assignment, and decide whether or not to re-compute that assignment when agents and rescue tasks are added to or removed from the environment. In addition, the algorithm uses less communication because it reduces the number of messages exchanged when a change is detected, and reduces the size of the messages sent. Also, message size and computation at each node is reduced by only computing and sending messages containing two values, as opposed to the entire domain of a variable. This reduction in computation begins to address our requirement of robustness to a rapidly changing environment, by reducing the amount of communication and computation used overall by the algorithm. In particular, we have conducted experiments with FMS, which show that it simultaneously provides a reduction of up to 99% in the total message size sent and of up to 99% in the total computation units needed (depending on the density of connections in a tree) compared to the standard max-sum algorithm. However, this comes at

the expense of the necessity of an amount of storage that is linear in the number of agents and the number of connections each agent has in the factor graph, as opposed to max-sum which requires no storage.

Branch-and-bound fast-max-sum (BnB FMS) (Section 4.3) — which builds upon FMS in order to specifically target our requirements of scalability and efficiency in communication (requirements 1 and 3) by giving it significant new capabilities: namely, an online pruning procedure that simplifies the problem, and a branch-and-bound technique that reduces the search space. This allows us to scale to problems to hundreds of tasks and agents. Our approach is two-fold: first, we present a novel, online pruning algorithm to reduce the set of tasks an agent can perform without effecting solution quality. Second, we introduce a new branch-and-bound technique that further reduces computation when computing fast-max-sum messages. Our empirical evaluation of BnB FMS shows that it finds solutions using up to 99% less computation and communication than FMS (which achieves the same utility as BnB FMS), and requires at most 49% fewer messages than branch-and-bound max-sum (BnB MS) in dynamic environments.

Bounded fast-max-sum (BFMS) (Section 4.4) — which gives FMS the new capability to provide bounded approximate solutions (addressing Requirement 6) on arbitrary graphs (i.e., not only trees), even in environments that change over time. This is achieved by removing dependencies between tasks and agents, according to how much impact they are likely to have on overall utility. In order to do this, we created the iterative GHS (iGHS) algorithm, which extends GHS (Gallager et al., 1983), a popular decentralised maximum spanning tree algorithm, to compute a maximum spanning tree on subsections of the constraint graph. This contributes to the state-of-the art by producing yet further reductions in communication and computation overheads over FMS, whilst still achieving good quality solutions. We have conducted experiments with BFMS which show reductions in communication and computation by up to 99% when compared to bounded max-sum (BMS), while obtaining within 1% of the optimal utility. The BFMS approach is general enough that it can be combined with the techniques from BnB FMS to completely meet our requirements of scalability, robustness, efficiency, adaptiveness, good quality solutions, and boundedness. However, the use of FMS, BnB FMS and BFMS does not entirely meet the robustness requirement. In particular, it is not completely decentralised, due to the computation distribution problem mentioned above, where the factor graph representation employed by FMS leads to an issue of which agents should compute and communicate for which tasks in the environment.

Spanning tree decentralised task distribution algorithm (ST-DTDA) (Section 5.4) — the first decentralised solution to the computation distribution problem (and so, the first decentralised solution to $R||C_{\max}$). In so doing, we use localised

message passing to specifically target scalability and robustness to agent failure whilst producing solutions of bounded quality (Requirements 1, 2 and 6). This is achieved through combining the use of heuristic-based spanning trees (for which we define 3 novel heuristics) and the min-max algorithm (which we show exhibits some useful properties that allow standard GDL computations to be significantly simplified). ST-DTDA operates in four steps, which are: (1) preprocessing, (2) building a heuristic-based spanning tree (ST), (3) running the min-max algorithm, and (4) computing the approximation ratio. In more detail, ST-DTDA first preprocesses the graph using a stochastic greedy algorithm as a starting point for finding solutions. Next, we introduce 3 novel heuristics for use when finding an ST-based approximation in order to to reduce complexity, then we use min-max, a localised message passing algorithm from the same family as max-sum to find a solution to the ST approximation. Next, min-max is used again in order to compute a worst-case bound on the error of the solution found previously, relative to the optimal solution — i.e., the biggest possible distance between the value of the optimal solution to the original problem, and the value of the solution found to the ST approximation. Finally, we combine the approximate solution value found in (3), a worst-case error bound, and a simple lower bound on the optimal solution value to estimate the approximation ratio of the solution found, relative to the optimal solution. Thus, in these four steps, ST-DTDA uses localised message passing through the min-max algorithm to find good quality, per-instance bounded approximate solutions in a distributed, efficient manner. Empirical evaluation of a number of heuristics for ST-DTDA shows solution quality achieved is above 90% of the optimal on sparse graphs, in the best case, whilst worst-case quality bounds can be estimated within 5% of the solution found, in the best case.

Given this, in Table 1.1, we explicitly show which of our requirements each contribution meets, and to what extent the requirement is met.

	FMS	BnB FMS	BFMS	ST-DTDA
Scalability	✓	✓✓	✓	✓✓
Robustness	✓	✓	✓	✓✓
Efficiency	✓✓	✓✓	✓	✓
Adaptiveness	✓✓	✓	✓	✓
Quality	✓	✓	✓	✓
Boundedness	×	×	✓✓	✓✓

TABLE 1.1: Outline of how our contributions relate to the requirements defined in Section 1.3. Here, we denote a requirement not satisfied by ×, and a requirement (strongly) satisfied by ✓(✓✓).

The work in this thesis has led to the publication of five papers:

1. S. D. Ramchurn, A. Farinelli, **K. S. Macarthur**, M. Polukarov and N. R. Jennings. Decentralised Coordination in RobocupRescue. *The Computer Journal* 53

(9): 1447–1461, 2010.

2. **K. S. Macarthur**, A. Farinelli, S. D. Ramchurn and N. R. Jennings. Efficient, Superstabilizing Decentralised Optimisation for Dynamic Task Allocation Environments. *The Third International Workshop on Optimisation in Multi-Agent Systems (OPTMAS)*, pages 25–32, 2010.

3. **K. S. Macarthur**, M. Vinyals, A. Farinelli, S. D. Ramchurn and N. R. Jennings. Decentralised Parallel Machine Scheduling for Multi-Agent Task Allocation. *The Fourth International Workshop on Optimisation in Multi-Agent Systems (OPTMAS)*, 2011.

4. **K. S. Macarthur**, R. Stranders, S. D. Ramchurn and N. R. Jennings. A Distributed Anytime Algorithm for Dynamic Task Allocation in Multi-Agent Systems. *The Twenty-Fifth Conference on Artificial Intelligence (AAAI)*, pages 701–706, 2011.

5. P. Scerri, B. Kannan, P. Velagapudi, **K. S. Macarthur**, P. Stone, M. E. Taylor, J. Dolan, A. Farinelli, A. Chapman, B. Dias and G. Kantor. Flood Disaster Mitigation: A Real-world Challenge Problem for Multi-Agent Unmanned Surface Vehicles. *The Autonomous Robots and Multirobot Systems Workshop (ARMS)*, 2011.

Papers 1, 2, and 4 form the basis of Chapter 4, paper 3 forms the basis of Chapter 5, and paper 5 discusses the application of dynamic task allocation algorithms such as ours to a novel exemplar dynamic task allocation application — specifically, using dynamic task allocation algorithms to coordinate a team of autonomous boats to deliver supplies to people effected by annual flooding in the Philippines.

1.5 Thesis Outline

The remainder of this thesis is organised as follows:

- In Chapter 2, we provide a review of related literature with respect to the requirements outlined in Section 1.3. We begin by providing an overview of commonly-used multi-agent coordination methods (Section 2.1), before going into more depth on our chosen area of DCOPs (Section 2.2). Here, we discuss the suitability of various methods of solving DCOPs, providing a breakdown of the anatomy of a DCOP solution algorithm. Next, in Section 2.3 we detail our chosen DCOP algorithms: max-sum, branch-and-bound max-sum, and bounded max-sum, and detail GHS, the minimum spanning tree algorithm used by BMS. Next, in Section 2.4, we

explain how some factor graph formulations that facilitate the application of max-sum can give rise to a problem we call *computation distribution*, and how this problem is analogous to an operations research problem called *scheduling on unrelated parallel machines*, or $R||C_{\max}$. Finally, we explain why the current state-of-the-art for $R||C_{\max}$ is unsuitable for use in a dynamic decentralised environment.

- In Chapter 3, we outline our chosen problem domain, give a detailed example of it, and formalise it as a factor graph. We then formulate the computation distribution problem that arises as a result of using this factor graph formulation.

- In Chapter 4, we present FMS: our first algorithm for reducing computation and communication in the max-sum algorithm. We also present BFMS and BnB FMS, which build on FMS to provide bounded approximate solutions, and to optimise performance on large-scale environments, respectively. Finally, we empirically evaluate the three algorithms.

- In Chapter 5, we present ST-DTDA, which is a distributed algorithm to quickly and efficiently solve the computation distribution problem which arises when formulating a factor graph as is needed to use FMS (i.e., with functions representing non-computational entities in the environment). We then evaluate this algorithm against existing benchmarks.

- In Chapter 6, we conclude and elaborate on future work.

Chapter 2

Literature Review

In this chapter, we begin by discussing a number of main approaches to designing multi-agent system-based task allocation approaches in Section 2.1: specifically, game theoretic techniques, auctions, MDPs, and DCOPs. Through this discussion, we clarify the decision to solve the chosen scenario as a DCOP. Against this background, we elaborate on existing methods used to solve DCOPs in Section 2.2. In particular, we study the anatomy of several key DCOP solution methods, and describe a number of approaches to coping with dynamism in the environment. Given this, in Section 2.3, we elaborate on the max-sum algorithm (Aji and McEliece, 2000), and two key extensions of it: BnB MS (Stranders et al., 2009) and BMS (Farinelli et al., 2009), show how they meet several of the requirements outlined in Section 1.3, and highlight where they do not. Next, we outline a problem which can arise from applying max-sum to certain representations of a task allocation problem — which we call *computation distribution* — in Section 2.4, and describe how current approaches to the problem do not meet our requirements. Finally, we provide a summary of our findings in Section 2.5.

2.1 Task Allocation Approaches

As discussed in Section 1.2, we believe that the multi-agent systems paradigm is a natural fit for dynamic task allocation problems. For example, disaster management scenarios involving the coordination of multiple autonomous decision-making entities representing different stakeholders, each with its own objectives. Fundamentally, coordination amongst these entities is central to the success of the solution mechanism, even when changes occur in the environment. Since the situation in task allocation scenarios such as disaster management is generally only perceived in part by each agent in the environment, the agents should work together to coordinate their actions and reallocate tasks when needed (Meissner et al., 2002).

In more detail, Waldock et al. (2008) define two levels of research into coordination

between agents in an MAS: explicit and implicit. Work into the former is concerned with decisions and plans being made jointly by all agents in the system, whereas the latter strand is focussed on agents making use of exchanging measurements and/or estimates such that they may perform cooperative inference, make local decisions, and/or develop an assignment strategy locally. In this work, we aim to place focus on explicit coordination, because because we wish to produce a solution in which there is no central point of failure (see Requirement 2), and adapt well to changes in the environment (Requirement 4).

Against this background, we have identified the four main groupings of multi-agent coordination techniques (all of which have been applied to task allocation): game theoretic techniques, auctions, MDPs, and DCOPs. In the remainder of this section, we discuss these groupings in further detail, while evaluating their applicability in dynamic task allocation problems (as per our requirements in Section 1.3).

2.1.1 Game Theory

Game theory is a branch of economics which centres around rule-based interactions between agents (Von Neumann and Morgenstern, 1944). These interactions are known as *games*, which consist of a number of players, a number of moves (known as strategies), and a payoff matrix, representing the payoff to each player for each combination of player strategies.

As mentioned earlier, there are two broad types of games: cooperative and noncooperative. In more detail, cooperative game theory involves a set of players which must form groups, known as *coalitions*, in order to take joint moves to maximise an overall joint payoff value. However, this payoff (known in cooperative game theory as *transferable utility*) must somehow be split amongst the players, trading off fairness and stability, which is another topic of research entirely (Von Neumann and Morgenstern, 1944). In contrast, noncooperative game theory studies a set of rational (selfish) agents, where the aim for each player is to choose the strategy which will give it the best payoff. Often, when each agent chooses its most beneficial strategy, it can result in the game reaching a state where none of the agents wishes to deviate from their current strategy, because they cannot possibly do any better; this is called a Nash equilibrium. To obtain such an equilibrium, the space of all possible combinations of strategies must be enumerated and then searched; a task which quickly becomes intractable with increasing numbers of players and strategies. In such settings, there are two types of strategies used by players: pure and mixed. The former explicitly define which action an agent will take in any given situation. Conversely, the latter assigns a probability distribution to each action, and so makes it difficult to assign the actions amongst agents. Thus, a pure strategy Nash equilibrium is where an equilibrium has been reached in a game where players only have pure strategies.

Of the many classes of noncooperative games that exist, we believe one of the most relevant to our work is that of potential games (Monderer and Shapley, 1996), which are also known as congestion games. This is because potential games are those in which a function (known as a potential function) of all agents' strategies can be used to express the incentive of all agents to change their strategy. That is, the potential function will increase if and only if an agent changing its strategy will lead to an increase in its own utility (payoff). Potential games are important because the existence of the potential function can be used to find pure strategy Nash equilibria, which are particularly desirable because they are not stochastic in nature. This means that, in these equilibria, it is explicit which action the player will take at any stage of a game, with any previous history of actions, and so finding equilibria becomes more tractable.

Game theory has been applied to a diverse range of topics, such as automated negotiation (Jennings et al., 2001), power management in sensor networks (Campos-Nanez et al., 2008), and task allocation (Chapman et al., 2009). In automated negotiation, Jennings et al. (2001) propose the use of game theoretic techniques when designing a protocol for use in agent interactions. Campos-Nanez et al. (2008) use game theory for power management in sensor networks by treating each sensor as a player in a game, with each possible action of a sensor treated as a strategy for its representative player. An application of game theory to our particular area of task allocation was presented by Chapman et al. (2009), and solved using their algorithm, overlapping potential game approximation (OPGA). In more detail, in Chapman et al. (2009)'s work, agents are considered to be the players in a game, and tasks they could be allocated to are the strategies. The problem is formulated as a Markov game, which is stochastic and therefore, in general, intractable due to the huge potential state space that must be searched when the outcome of actions is not finite. Hence, the global utility function is approximated such that the game can be approximated into a series of static potential games, with complete information. A distributed stochastic algorithm is then used by agents individually to find equilibria in these games. However, since OPGA uses such a distributed stochastic algorithm, it can only find a local maximum, as opposed to the global maximum that we wish to aim to optimise. Moreover, Chapman et al. assume that the assignment of agents to tasks is static, in that the set of tasks does not change within the current allocation. If the set of tasks were to change, the agents would have to recompute the entire allocation again. Finally, the allocation mechanism employed by OPGA ignores how different coalitions form based on when agents arrive at a task (i.e, the synergistic effect that can be exhibited by specific groupings of agents). Therefore, due to this, and the fact that there are no quality guarantees (Requirement 6), this approach does not fit our specific requirements. Moreover, they assume linear utility functions, which are too restrictive for our model, which is general enough that utility functions can be arbitrary.

Against this background, we have chosen not to use game theoretic mechanisms to

model our scenario. This is because both noncooperative and cooperative game theory are not a natural fit for dynamic task allocation. In more detail, as mentioned earlier, in collaborative systems like ours, cooperative game theory requires the additional computational complexity of distributing joint rewards amongst agents on top of the complexity of finding a solution, which will make our requirements of scalability, robustness and adaptiveness (see Requirements 1, 2 and 4) more difficult to achieve. The application of both cooperative and noncooperative game theory would require creating an approximation of our problem involving selfish agents. In addition to this, finding the Nash equilibrium does not necessarily equate to finding the global maximum: it can be a local maximum, and in dynamic task allocation we are concerned with finding a global maximum. Both of these problems with noncooperative game theory will impact our requirement of quality (see Requirement 5).

2.1.2 Auctions

Auctions are a popular method used to achieve multi-agent coordination. Auctions typically consider actors to be rational, competitive agents (i.e. selfish agents) placing bids on items being sold by other agents. There exist a number of applications of such techniques for coordination of agents in disaster management. In particular, López et al. (2005, 2008) presented an auction system called MASICTUS[1] for coordinating ambulances and neurologists from different ambulance trusts in cases where people have strokes. The system conducts auctions using real money to allocate the ambulances that cost the least money, and are near to the emergency. MASICTUS is reactive to changes in its environment even whilst an ambulance is on its way to a hospital, which is important for an application in such an uncertain environment. The system relies on human interaction, in that patient and ambulance information must be entered by a human, and decisions must ultimately be made by human too. The autonomy in this system lies in the decision support system that it provides: it uses autonomous agents to model ambulance trusts in order to allocate ambulances, once the ambulances and their preferences have been input by a human. As is typical in auction domains, the agents in MASICTUS are competitive and self-interested, unlike our domain. In addition to this, MASICTUS is not distributed across computers and so will have a single point of failure: the computer upon which it is located.

Another application relating auction mechanisms to disaster management is that presented by Berhault et al. (2003). In this mechanism, combinatorial auctions are used to find a schedule of which target areas to visit for a number of robots. A combinatorial auction is one in which bidders place their bids on a combination of items in order to win them. Such auctions are appropriate for scheduling problems, where each agent

[1] A portmanteau of MAS (multi-agent system) and *ictus*, a medical term referring to a sudden event such as a seizure, collapse, faint, or, as in this case, an acute cerebrovascular accident (more commonly known as a stroke).

must choose a number of targets to visit, and an order in which to visit them. However, this mechanism does not apply to our specific problem because the complexity involved in calculating the winning bids for each possible combination of goods means that approximate solutions must be found.[2] This complexity leads to difficulty in adapting to changes in the environment (Requirement 4), as each time a change is made, the bid calculation will have to be repeated at each agent, which is likely to be computationally expensive.

In terms of single-item auctions, work by Lagoudakis et al. (2004) into single-item auctions for multi-agent coordination showed that the problem of finding an optimal allocation of agents to arbitrary targets is NP hard, even when the environment is entirely known to each agent. To address this, they presented the PRIM ALLOCATION algorithm, which uses a single-item auction mechanism to minimise total travel costs when allocating agents to their targets. PRIM ALLOCATION provides guarantees on the quality of the allocations it produces (which satisfies Requirement 6), specifically that the total cost an allocation will incur is at most twice as much as the total cost of the optimal allocation. Whilst this method provides close-to-optimal allocations as opposed to optimal ones, it benefits from simplicity and robustness (satisfying Requirement 2), as well as fast communication, adaptiveness (satisfying Requirement 4), and efficiency (satisfying Requirement 3). The total number of bid messages sent by this method is around 5% of that needed for the combinatorial auctions discussed previously (Berhault et al., 2003). However, PRIM ALLOCATION is inefficient in that it must operate over a complete graph, where all agents have a cost to do all tasks, even though agents cannot reach all tasks (i.e. some have an infinite cost), instead of exploiting sparsity. This means that PRIM ALLOCATION is unlikely to meet our scalability requirement (Requirement 1).

Given this background, we can see that auctions could be applicable to our problem, but are not a natural fit, since the use of market-based mechanisms to coordinate agents introduces another problem of how to represent and distribute transferable utility amongst buyer and seller agents. In addition to this, communication of item bids could lead to slow decision-making if done over an unreliable communication line, impacting scalability, robustness and adaptiveness (Requirements 1, 2 and 4). Thus, we have chosen not to further investigate auction mechanisms for dynamic task allocation.

2.1.3 Markov Decision Processes

MDP approaches involve the use of various different families of MDPs, including POMDPs and MTDPs, which are used for modelling sequential decision making under various degrees of uncertainty. The solution to a problem belonging to any family of MDPs is called an optimal policy, and is a mapping from each potential world state to the best

[2]Determining the winning bid in this case is NP complete (Berhault et al., 2003).

action to take for a given agent: essentially a lookup table of actions, indexed by world state. Finding such a policy is often computationally complex and time consuming, and so, approximations often have to be identified as opposed to full solutions (Nair et al., 2004). To optimise this policy, a mapping which maximises the reward gained must be found.

In more detail, an MDP is a model of a scenario containing a single agent which has full observability over its environment. It should be noted that full observability means that the agent will be able to observe the new state of the environment immediately after an action has been performed in it. Despite this full observability, the best known bounds for finding an optimal MDP policy have been proved by Papadimitriou and Tsitsiklis (1987) to be P Complete.[3] In contrast, a POMDP is an MDP which is only partially observable: that is, changes resulting from making actions are not always observed accurately. This uncertainty means that the worst-case computational complexity of finding a solution to a POMDP is PSPACE hard[4] (Becker et al., 2003). Now, an MTDP, defined by Pynadath and Tambe (2002), extends the POMDP approach to apply it to teams of agents working toward a common goal in a partially observable environment. In a POMDP approach, each agent in the team needs to be represented individually; the MTDP approach improves this by considering the joint action space of all agents (i.e. all possible combinations of actions to be taken by agents) in a single representation. However, the MDP family of models presents a centralised solution to a multi-agent problem; the entire system is modelled as a number of MDPs/POMDPs or a single MTDP and solved in a single place, before being communicated to agents.

A number of MDP-based mechanisms have been defined for use in decentralised environments such as ours; for example, the MMDP (Boutilier, 1996), decentralised Markov decision process (DecMDP) (Becker et al., 2003) and collaborative multi-agent factored Markov decision process (Guestrin, 2003) were designed with decentralised control in mind. Whilst these methods do allow agents to maximise global utility, as opposed to individual utility, modelling of the system as an MDP must still be done for each change to the environment and projected utility values. This is potentially very time consuming, since the search space the algorithm must cover each time is exponential, and so is unrealistic for our problem, especially in large environments. Thus, MDP approaches violate Requirements 1, 2 and 4 (scalability, robustness and adaptiveness), and so, we opt not to use MDPs in our work.

[3] Informally, P Complete is the set of the hardest problems in the set of problems solvable by a Turing machine in polynomial time.

[4] Informally, a problem is PSPACE hard if it is *at least* as hard as the hardest problems in PSPACE — the set of problems that are solvable in polynomial space by a Turing Machine.

2.1.4 Distributed Constraint Optimisation Problems

A DCOP consists of agents, variables and functions, which act as constraints on variable values. In more detail, each agent controls one variable, which has a domain of potential values. Functions constitute mappings from variable states to utilities, and so, form constraints on which values those variables may take. Finding a solution to a DCOP is equivalent to finding an assignment from each variable to a value, such that a global utility function composed from all the local functions is optimised. In general, these solutions are found by arranging the variables and functions into a graphical representation, which facilitates the exploitation of what are known as *sparse graphs*. In more detail, in such graphs, each agent only needs to interact with a few other agents local to it, allowing the decomposition of a global payoff function into a sum of payoffs gleaned from each of these local interactions. Some applications of DCOPs are meeting scheduling (Maheswaran et al., 2004), task allocation (Scerri et al., 2005), and mobile sensing (Stranders et al., 2009).

Now, many decentralised methods for solving or at least approximating solutions for DCOPs exist, and have been defined in the literature. Such methods can largely be divided into two groups: *complete algorithms* such as ADOPT (Modi et al., 2005) and DPOP (Petcu and Faltings, 2004), and *approximate algorithms* such as BMS (Farinelli et al., 2009), divide and coordinate subgradient algorithm (DaCSA) (Vinyals et al., 2010) and distributed asynchronous local optimization (DALO) (Kiekintveld et al., 2010). In more detail, complete algorithms find optimal solutions to DCOPs, but usually incur large communication and computation overheads to do so. These overheads often grow exponentially (Petcu and Faltings, 2004), which impacts scalability in the number of agents in the environment (Requirement 1). On the other hand, as their name suggests, approximate algorithms cannot guarantee they will return the optimal solution, but can often give quality guarantees on the solutions they provide (Farinelli et al., 2009; Vinyals et al., 2010; Kiekintveld et al., 2010). In doing this, approximate algorithms incur smaller overheads than the complete algorithms, which then allows them to scale more easily in the number of agents (Requirement 1), be more efficient in their communication (Requirement 3), and potentially adapt to changes in the environment (Requirement 4).

We have chosen to investigate DCOP formulations and algorithms in our work, because, as we show in the next section, they allow our scenario to be modelled intuitively, and DCOP algorithms have the potential to meet all of the requirements we laid out in Section 1.3. In more detail, while finding optimal solutions to DCOPs can be very complex, especially in large problems, we have found a number of decentralised (Requirement 2) approximate algorithms which provide quality guarantees on their solutions (Farinelli et al., 2009; Vinyals et al., 2010; Kiekintveld et al., 2010) (Requirement 6). In doing so, these algorithms incur fewer overheads than the other coordination approaches, which makes them scalable (Requirement 1), more likely to adapt to changes in the envi-

ronment (Requirement 4) and more efficient in terms of communication (Requirement 3). Given this, in the next section, we break down a number of key DCOP solution algorithms and assess them for our purpose.

2.2 Decentralised DCOP Solution Methods

As mentioned in Section 2.1.4, a solution to a DCOP is a set of mappings from each variable to an element in its domain, such that some global objective function is optimised. Finding such a solution could be done by enumerating all possible combinations of variables and values until a combination is found which optimises the function. However, this search space will be very large in a system with many agents, each with large domains of possible actions, and, thus, many values that the function could return. Hence, such a search will quickly become intractable.

Against this background, in this section, we analyse the main algorithms for solving DCOPs. Specifically, we focus on: DPOP (Petcu and Faltings, 2005a), ADOPT (Modi et al., 2005), DaCSA (Vinyals et al., 2010), DALO (Kiekintveld et al., 2010), and the GDL family of algorithms (Aji and McEliece, 2000): specifically, max-sum. These algorithms can broadly be split into two groups: complete algorithms (DTREE/DPOP, ADOPT) and approximate algorithms with guarantees (DaCSA and DALO). However, max-sum and its variants, BnB MS (Stranders et al., 2009) and BMS (Farinelli et al., 2009), fit into both groups — if applied on an acyclic graph, then max-sum is complete, otherwise, BMS will provide a bounded approximation. By way of an introduction, we give a brief overview of each algorithm below:

DTREE/DPOP — distributed pseudotree optimisation procedure (DPOP) (Petcu and Faltings, 2005a) is an extension to DTREE (Petcu and Faltings, 2004), and uses depth-first search (DFS) in order to find optimal solutions on arbitrary graph structures, at the expense of large messages. Superstabilizing distributed pseudotree optimisation procedure (SDPOP) (Petcu and Faltings, 2005b), an extension of DPOP, is so far the only one of our main DCOP algorithms to specifically target domains that change over time, by using a dynamic DFS tree algorithm. However, in doing this, SDPOP has a long convergence time and has the same issue with large messages that DPOP has.

ADOPT — asynchronous distributed constraint optimisation (ADOPT) (Modi et al., 2005) adapts the asynchronous backtracking (ABT) algorithm (see Yokoo et al. (1992)) in order to apply it to finding solutions to DCOPs. This is done using a technique called iterative thresholding, which will be explained in more detail in Section 2.2.3, and, whilst only needing a polynomial amount of memory, does send a large number of messages.

DaCSA — divide and coordinate subgradient algorithm (DaCSA) (Vinyals et al., 2010) is an anytime, bounded algorithm with quality guarantees. The general principle of the algorithm is to divide the DCOP up into a number of smaller local problems and solve them asynchronously. This reduces communication and computation whilst producing guaranteed good quality solutions. However, the algorithm does not specifically cater for changes in the environment, and so, would need to be re-run after a change.

DALO — distributed asynchronous local optimization (DALO) (Kiekintveld et al., 2010) uses the principles of k-size optimality (Pearce et al., 2006) and t-distance optimality (Yin et al., 2009) to provide anytime local approximations with quality guarantees. This is done by starting with a random variable assignment and monotonically improving it over time. However, DALO is partially centralised in that it relies on a number of agents to make decisions and send them out locally. In addition to this, DALO does not cater for changes in the environment, and would instead need to be re-run after a change.

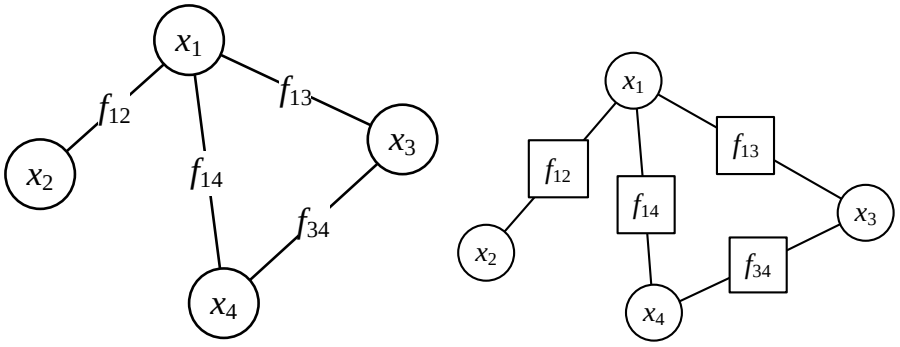
Max-sum — Max-sum is a member of the GDL (Aji and McEliece, 2000) family of algorithms, which also includes algorithms such as the sum-product and max-product algorithms, which have been shown to be applicable to graphical probabilistic models (see Kschischang et al. (2001); Wainwright et al. (2004)), and information theory (Mackay, 2003). All algorithms in the GDL family will find an optimal solution for any problem which can be modelled as an acyclic graph. Max-sum can be sped up through the use of a number of computation saving techniques, as shown by the BnB MS algorithm (Stranders et al., 2009). In addition to this, in graphs with cycles, the BMS algorithm (Farinelli et al., 2009) can be used to find bounded approximate solutions.

Given this, in the remainder of this section, we first compare the graphical representations used by the algorithms (Section 2.2.1), before assessing the algorithms against our requirements in Section 1.3, by breaking the approaches down into three logical phases which are exhibited by all of the algorithms: preprocessing (Section 2.2.2), message passing (Section 2.2.3) and solution computation (Section 2.2.4). Following that, we detail a number of important properties to consider when developing algorithms for dynamic environments (Section 2.2.5), in order to inform our decisions when developing our own algorithms. We then discuss our findings in Section 2.2.6.

2.2.1 Representation of Problem

All of the DCOP algorithms we consider rely on specific graphical representations of problems in order to operate. In particular, the representations used by the algorithms we study are: coordination graphs (DPOP, ADOPT, DaCSA, DALO), and factor graphs

(max-sum). We give examples of each type of graph in Figure 2.1 and describe them below.



(a) Coordination Graph. (b) Factor Graph.

FIGURE 2.1: Examples of each representation of the environment used by DPOP, ADOPT, DaCSA, DALO and max-sum. Both graphs show the function $F(x_1, x_2, x_3, x_4) = f_{12}(x_1, x_2) + f_{14}(x_1, x_4) + f_{13}(x_1, x_3) + f_{34}(x_3, x_4)$

Coordination graphs (sometimes known as constraint graphs/networks), used by DPOP, ADOPT, DaCSA and DALO, and shown in Figure 2.1 (a), use nodes to represent variables (agents) in the global objective function, and edges to represent dependencies between variables (shown as circles containing x_1, x_2, x_3, and x_4 in Figure 2.1 (a)). In more detail, each edge represents a local payoff function (see f_{12}, f_{14} etc. in Figure 2.1 (a)), which can be used to calculate a value for each possible combination of actions from agents connected by a local dependency. In this case, the decomposition of the problem is such that the global payoff value is the sum of all local payoffs (i.e. the sum of payoffs over each edge in the graph).

Now, max-sum uses a slightly different representation called a factor graph, shown in Figure 2.1 (b), which uses different types of nodes to represent variables (agents) and local payoff functions. In more detail, a factor graph is a bipartite undirected graph, and is appropriate for use in problems where the global payoff function can be factored into a number of local functions, each of which will be represented by what is known as a factor node in the graph (denoted f_{12}, f_{13}, f_{14}, f_{34} in Figure 2.1 (b)). Each variable in the global payoff function is represented by a variable node in the graph (denoted x_1, x_2 and x_3 in Figure 2.1 (b)). An edge exists between a factor node and a variable node if and only if that variable is an argument to the function represented by that factor node.

These representations do not have any bearing on whether or not the algorithms meet our requirements. However, both representations do have the benefit of being flexible representations of an environment: nodes can be added and removed without requiring the entire re-modelling of the system. Given these representations, we can now go on to compare the preprocessing phases of the algorithms we study.

2.2.2 Phase 1: Preprocessing

All of the algorithms we consider involve a preprocessing phase in order to prepare the graph they are given for the message passing phase. In general, the preprocessing phase involves building a spanning tree[5] of the graph in order to simplify communication and computation. In more detail, cyclic factor graphs and coordination graphs cause unneeded communication and computation overheads in GDL algorithms because agents can get stuck sending messages in an infinite loop — limited theoretical results exist proving GDL convergence on cyclic graphs (Wainwright et al., 2004). As such, it is preferable to run algorithms on tree-structured graphs, and thus, a preprocessing stage is needed to convert cyclic graphs into acyclic graphs.

The complete algorithms (ADOPT and DPOP) use this phase to ascertain a root node in the graph. This root node is the eventual source of decision making, as we will explain in Sections 2.2.3 and 2.2.4. This centralises the algorithms, as it introduces a single point of failure: if the root node were to be destroyed or incapacitated then the entire algorithm, including the preprocessing phase, would have to be run again.

The easiest method to build a tree of a coordination graph (and that used by ADOPT) is to use DFS. In particular, the output of DFS execution is what is known as a DFS tree, which must always contain nodes (agents) which are adjacent in the original graph in the same branch. This ensures relative independence of nodes in different branches of the tree, allowing parallel search, after which results can be combined (Petcu and Faltings, 2007). For any given coordination graph, a DFS tree is considered 'valid' if no payoff functions between agents in different subtrees of the DFS tree exist.

Once the DFS tree has been generated, each node is aware of its parent node and child nodes. The root node is one which has no parent node. Each node will also be aware of its neighbouring nodes, i.e., those that are not parent or child nodes but were connected to the node in the original graph. Thus, the cycles in the original graph are preserved within the nodes, but a strict tree structure is in place to restrict message flow.

Now, whilst this DFS tree is enough for ADOPT to operate, DPOP requires more information to be encoded in the graph. In more detail, DPOP uses the DFS algorithm to turn the DFS tree into what is known as a pseudotree. A pseudotree, first used for constraint satisfaction problem (CSP) solutions in Freuder and Quinn (1985), is a DFS tree in which agents are aware of more than just their parents and children. More specifically, a pseudotree consists of nodes, tree edges, and back edges, where the latter represent cycles in the coordination graph (i.e., links between neighbour nodes). Each node has a single parent, and multiple children, pseudoparents, and pseudochildren; with pseudoparents and pseudochildren being those attached to the node by back edges.

[5] A spanning tree is a connected cycle-free subgraph with the same vertex set as the original graph. The spanning tree provides a unique graph between any two nodes, removing any cycles.

The approximate algorithms we study take a different approach to tree formation. In more detail, DaCSA and DALO split the overall graph into smaller subgraphs, and form a spanning tree of each subgraph. Thus, DALO is given a value for either t or k when it is executed.[6] This value is used in order to find all of the possible groups of agents, relative to the value of t or k, that can be formed in the graph. A leader agent must then be established for each of these groups, and each agent's constraint table (i.e., the value of each state it can take, with respect to its own variable value and those of its neighbours). Each agent may be a member of numerous groups at once, so it is required for them to flood their constraint table far enough for all of their potential group leaders to receive their information.

DaCSA takes a similar approach to DALO, in that the agents split the graph up into smaller, easier to manage graphs (called *subproblems* in DaCSA). However, DaCSA differs both in the graph division technique and the algorithm execution. Firstly, DaCSA uses a popular linear programming solution technique, called Lagrangian relaxation, to divide the graph.[7] After this phase, there exists a series of subproblems: one for each variable in the original problem, along with its neighbouring variables. As such, the neighbour variables present in each subproblem can and will overlap (i.e., exist in more than one subproblem), and so, at this point in the algorithm, the variable value assignments may not be consistent between subproblems. Secondly, DaCSA differs from DALO in its execution because DaCSA involves interleaving the preprocessing and message passing phases at each node. Hence, each time this preprocessing phase is run, the variables modify their subproblems according to some variable parameters (updated during the message passing phase), in order to reach consistent variable assignments.

Now, where DPOP, ADOPT, DALO and DaCSA are optimal algorithms, max-sum is not strictly an optimal or approximate algorithm, because it is only optimal on tree structured graphs. Max-sum does not re-structure the factor graph during its preprocessing phase. Instead, the preprocessing phase consists only of each agent initialising its variable with a randomly generated value. This initial value ensures that there is always a single best value to choose for each variable. This is essential to the execution of max-sum as it avoids a phenomenon known as graph symmetry, where an arbitrary decision would have to be made between two values with identical utilities, potentially ending up in conflicting solutions at different agents.

Stranders et al.'s work on BnB MS (Stranders et al., 2009) reduces the computation done by max-sum in phase 2 (see Section 2.2.3) through the addition of a preprocessing phase to max-sum, which they call *domain pruning*. In more detail, the domain pruning

[6] t is from t-distance optimality (Yin et al., 2009). t controls the size of groups as each group will contain one central node plus all nodes up to and including t hops from it. k is from k-size optimality (Pearce et al., 2006). k is the fixed group size: each group must contain k connected agents.

[7] In more detail, Lagrangian relaxation involves writing the most important constraints into the objective function (and thus, removing edges in the coordination graph) in order to ensure that they are satisfied. By removing edges, this ensures the resulting local graphs are acyclic.

step uses message passing to find which elements in each variable's domain are definitely not in the optimal solution, and removes them, so as to reduce the state space that needs to be explored at each function node in the message passing phase. We elaborate on both BnB MS and domain pruning in Section 2.3.3, as we use BnB MS as a starting point to reduce computation in one of our own algorithms (see Section 4.3.1).

As mentioned earlier, max-sum is only provably optimal on acyclic (tree-structured) factor graphs. To combat this, some work has been done into preprocessing for removing or avoiding cycles in factor graphs, which could be used for max-sum. In particular, Kschischang et al. (2001) discuss a number of methods for removing or avoiding cycles: clustering and stretching variable nodes. The clustering approach involves replacing a number of nodes in the graph with one 'cluster' node, which must maintain connection to all nodes with connections to those in the cluster. Whilst this will eliminate cycles if done correctly, it will increase the size of messages sent between clusters, as a message representing all combinations of states of nodes within a cluster will need to be sent. Stretching a variable node involves representing that variable over a number of other nodes in the graph. Thus, instead of just immediate neighbours having a given variable in their payoff functions, neighbours of those neighbours will also consider it. This allows that variable's node to be removed from the graph, cutting any cycles that it may be involved in, whilst maintaining that variable's effects on the graph. While these approaches are attractive because they remove cycles, they exponentially increase the complexity of functions and/or variable domains (Kschischang et al., 2001).

Another, more feasible, approach used to apply max-sum to graphs with cycles is BMS (Rogers et al., 2011), which was developed as an extension to max-sum, which provides bounded approximate solutions on cyclic factor graphs. In more detail, BMS's preprocessing phase involves running the GHS (Gallager et al., 1983) algorithm over the nodes in the factor graph in order to find the maximum spanning tree of the graph. We elaborate on this process in Section 2.3, as we use a similar technique in our algorithm (see Section 4.4.2) because the GHS algorithm scales well in the number of agents.

We next detail the second phase of DCOP solution algorithms, which consists of agents exchanging messages about the utility of variable assignments in order to find a good quality (or optimal) solution to the DCOP.

2.2.3 Phase 2: Message Passing

The message passing phase of DCOP solution algorithms involves agents exchanging messages along edges in the preprocessed graph produced in phase 1, if one was produced, in order to find the best solution to the DCOP. In more detail, in this phase, agents communicate information about the utility they would gain for different values of their variable(s) in order to collaboratively work out the most beneficial combination

of variable values.

Now, there are three main methods to find the best solution used by the algorithms in this phase: dynamic programming, distributed search, and distributed backtracking. More specifically, the dynamic programming approach is used by DPOP, DaCSA and max-sum, distributed search is used by DALO, and distributed backtracking is used by ADOPT.

In more detail, DPOP, DaCSA and max-sum rely on agents explicitly sending utility functions and values to their neighbours, in order to propagate the global utility about the graph. DPOP uses the most inefficient method of doing this, requiring the agents to propagate their utility functions from the leaves of the tree to the root, in messages known as UTIL messages. These UTIL messages grow in size as they move toward the root, as each node will combine its own utility function with that of its children. This reliance on the root node to finally collate the utility functions into a global utility function is what makes DPOP inherently centralised — if, at any point, were the root node to fail, this phase of the algorithm would not complete. Now, while DPOP sends a linear number of messages in this phase, their size is space-exponential in the induced width[8] of the graph, and a lot of computation must be done at each node. Thus, the UTIL messages produced by DPOP can grow very large with the number of back edges in the pseudotree — since back edges represent cycles in the constraint graph, there is a direct correlation here between cycles in the constraint graph and the size of UTIL messages.

In the DaCSA algorithm, each agent informs its neighbours (with whom it shares variables) of the assignment of its local variable(s). Thus, when an agent receives a message it can update the parameters it stores for each variable in order to reduce disagreements in assignments the next time the preprocessing phase is run. This must be done so that the agents do not have conflicting views on which values the variables related to them are taking.

In max-sum, nodes will only send relevant utility information to their neighbours, as opposed to utility information for all possible (and potentially unrelated) variable values in the system. In more detail, the message passing phase in max-sum uses two types of messages: from variable node to factor node, and from factor node to variable node. These messages, flowing into and out of variable nodes in the factor graph, communicate the total system utility for each state of each variable. The calculation of these messages can prove computationally expensive in dense graphs, however, because each function must explore the entire joint state space of its variables when computing messages. This

[8] By *induced width*, Petcu and Faltings (2005b) refer to Dechter and Cohen (2003)'s method of finding the induced width of a graph, which involves recursively expanding all possible parents of each node in the order of a DFS tree formation of the graph, to form an *induced graph*. Once this has been done, the maximum number of parents of any node in this induced graph is taken to be the induced width of that graph.

can be simplified through the use of BnB MS (Stranders et al., 2009), which introduces search trees and branch-and-bound in this message passing step to reduce the amount of the state space that needs to be explored at each function node — we elaborate further on BnB MS in Section 2.3.3. Nevertheless, regardless of how the function computation is performed, at any time during the propagation of these messages, an agent can calculate which state it should be in by maximising over the messages it has received. We give more detail on max-sum in Section 2.3.2.

On a tree-structured graph, the max-sum messages are guaranteed to converge to an optimal solution, such that each agent knows which value its variable should take. However, on a cyclic graph, these messages are not guaranteed to converge to a variable assignment. Hence, in BMS, max-sum is now run on the minimum/maximum spanning tree (MST) produced by the preprocessing phase of BMS. BMS does not terminate here, as a solution passing phase must be employed in order to calculate how good the approximation gained by BMS is (see Section 2.3.5 for more details).

In contrast to the above algorithms, DALO takes a decentralised search-based approach to distributing utility functions: essentially, nodes send out potential assignments of variables, which are changed as and when better solutions can be found. In DALO, leader nodes work out the best assignment for the variables in their group by using a centralised variable elimination algorithm, like in DPOP. As variables can be a part of many different groups, variable assignments could change at any time, as the leader nodes of each group make their decisions asynchronously. This changeability of variable assignments is handled by ensuring that each variable broadcasts a message to its group leaders informing them when its assignment has changed. Optimisation at the leader nodes is then started again. This iterative process continues until the values converge at each leader node (i.e., the leader node no longer receives any messages).

Finally, ADOPT takes a distributed backtracking (Yokoo et al., 1992) approach to finding a DCOP solution. Agents in ADOPT attempt to find the optimal solution by systematically testing different combinations of variable values, whilst maintaining upper and lower bounds on the cost of a solution. Then, if an agent finds the cost of a given assignment is too high with respect to the bounds it has, it will instigate a *backtracking* process. This process essentially disassembles the current solution far enough to allow the agents to try different variable value combinations. This search process is known as *best-first* search, and has the problem of needing to repeatedly reconstruct solutions when backtracking. BnB-ADOPT (Yeoh et al., 2010) addresses this problem by using branch-and-bound depth-first search instead, which reduces the number of times solutions will need to be reconstructed. However, even with this improvement, the choice between using ADOPT and using BnB-ADOPT is domain-specific, since in some domains BnB-ADOPT uses more communication and computation than ADOPT. Recently, a hybrid of the two algorithms was presented, called ADOPT(k) (Gutierrez et al., 2011) — this algorithm uses a parameter, k to control how much the algorithm

behaves like ADOPT (at $k = 0$) or BnB-ADOPT (at $k = \infty$). However, as yet there is no automated way of choosing the value of k, since the optimal value of k is domain-specific.

At the end of this phase, ADOPT and max-sum have terminated, with each agent aware of which value is the most beneficial for its variable to take. DPOP, DaCSA, BMS and DALO, however, require a third phase in order to propagate the solution to all agents.

2.2.4 Phase 3: Solution Computation

In phase 3, DPOP and DALO distribute computed solutions, while BMS and DaCSA calculate approximation ratios. In more detail, DPOP and DALO use the solution computation phase to let the root of the graph/group leaders to distribute their computed solution throughout the graph.

In DPOP, the root node sends a VALUE message to its children informing them of its optimal variable assignment (i.e., the best value it can take). The remainder of the graph then propagates further VALUE messages, informing their children of their own optimal variable assignment with respect to earlier ones. We consider this to be a centralised approach, as if any node were to be removed from the graph during execution, then the algorithm would have to re-run to find a solution.

Similarly, DALO uses this phase for the group leaders to distribute their variable allocations to their group members. This must be done monotonically in order to avoid conflicting assignments. This is done using a locking mechanism similar to that used for mutual exclusion in distributed computing applications (Dijkstra, 1968).

In contrast, BMS and DaCSA use this phase in order to calculate bounds on their approximation. In order to do this, DaCSA must calculate the utility of the current possible solution. As the entire assignment is never all held in one place, DaCSA aggregates the assignment at the same time as the utility of the solution. This is done by having each node aggregate the solution information from its children, and send the information to its parent node. The node will then wait for a reply from its parent, carrying the true solution value, thus allowing each node to calculate the approximation ratio.

BMS takes a very similar approach to DaCSA, in that it also involves gathering up the complete assignment and solution value by message propagation. In this phase, BMS also aggregates the total weight of edges removed, and thus, the utility lost. This can then be combined with the solution value and used to calculate the approximation ratio of the solution (see Section 2.3.5 for more details). Thus, BMS has been shown to produce solutions typically within 95% of the optimal solution on randomly generated scenarios (Farinelli et al., 2009).

Now, in order for any of the discussed DCOP solution algorithms to satisfy our Requirements laid out in Section 1.3, they must be able to cope in an environment which

changes over time. In more detail, while the SDPOP extension of DPOP specifically caters for such environments through the use of a dynamic DFS algorithm, ADOPT, DaCSA, DALO and max-sum have no special procedures for when a change in the environment (and so, the graphical representation of their environment) occurs, and so, the obvious way for them to deal with such situations is for them to be completely run again from scratch in order to find a new allocation of variables to values. This can incur communication and computation which is not necessary: for example, a small change could result in all agents needing to re-send their messages and re-compute only to find that their new assignment is no different to their assignment before the change happened. Hence, special provision must be made for the algorithms to be used in environments that change over time, and so, in the remainder of this section, we detail key concepts needed to evaluate algorithms with the properties we wish to introduce.

2.2.5 Coping with Dynamism

A number of key theoretical properties of algorithms are used in the literature in order to characterise the performance of DCOP algorithms in dynamic environments, with respect to the requirements we presented in Section 1.3 (and, in particular, requirement 4). Specifically, these properties are self- and superstabilization, anytime algorithms and competitive analysis.

2.2.5.1 Self- and Superstabilization

When designing a distributed algorithm for an environment that changes over time, it is important that the algorithm is self-stabilizing (Dijkstra, 1974). In more detail, a self-stabilizing algorithm is distributed across multiple agents and able to return those agents to some legitimate state after a fault or change has occurred. More specifically, when some state that invalidates a pre-defined legitimacy predicate is reached, the algorithm must be able to return all agents to a legitimate state in a finite amount of time, and in a distributed fashion. In our dynamic task allocation context, this legitimacy predicate is equivalent to a state where all agents are assigned to a task in order to maximise some overall utility function. Dolev and Herman also introduced the concept of superstabilization, which builds upon this, but ensures that the algorithm does not enter an unsafe state during recovery (Dolev and Herman, 1997).

In more detail, in addition to Dijkstra's legitimacy predicate, which must hold before and after a fault, Dolev and Herman introduce a passage predicate, which must hold during the recovery from that fault. Thus, illegitimate states are partitioned into two groups: those that satisfy the passage predicate, and those that do not. In a superstabilizing system, states which are both illegitimate and do not satisfy the passage predicate must never be reached. Now, as the passage predicate must hold between two legitimate

states (before and after a fault), it must be weaker than the legitimacy predicate, but must still ensure that the system remains in an acceptable state when the legitimacy predicate does not hold.

We can put this notion of a *fault* in the system in the context of our dynamic task allocation problem depicted in Section 1.1, where we have an environment filled with agents and tasks, where each agent can do a subset of the tasks. Against this, we can define a fault as being a change in that environment — an agent/task being added, an agent/task being removed, or a change in the set of tasks an agent can do.

Given this, it is clear that self-stabilizing algorithms have applications in the dynamic task allocation domain, since they ensure that agents are eventually assigned to tasks after a change in the environment, and that they don't do anything untoward whilst they react to the change. Thus, constraint optimisation algorithms used to find the optimal allocation of agents to tasks must be able to recover from changes in the environment in a timely manner. Now, whilst self-stabilization does ensure that a system responds to a fault in a sensible manner, and within finite time, it does not ensure any properties of safety. This means that during the time between the fault and the legitimate state found following recovery, the system can hold a state which could act to its detriment. For example, if a number of agents had been assigned to tasks, and a task was removed, the assignments of the agents could be removed during reallocation, thus wasting time until the recovery has completed. Therefore, superstabilizing constraint optimisation algorithms must be employed in order to preserve these safety properties. For example, in our problem, the allocation of agents to tasks should not be cleared and re-calculated after a fault (such as a building falling down, or an agent being incapacitated), as this would lead to agents stopping what they are doing until a new allocation has been computed — if, for example, a medical agent were to abandon a patient mid-procedure, this could be disastrous. Instead, a new solution must be found without unneeded disruption to the tasks that the agents were carrying out before the fault occurred.

The only current superstabilizing algorithm for solving DCOPs in changing environments is SDPOP (Petcu and Faltings, 2005b), which extends the DPOP algorithm (Petcu and Faltings, 2005a) in order to make it tolerant to faults without the need to recompute the entire communication structure of the agents, and the allocation. However, as with DPOP, SDPOP sends a linear number of space-exponential messages, and so is not practical. In addition, as SDPOP is optimal, the legitimacy predicate in SDPOP ensures that all variables are assigned to values which maximise aggregate utility over the system. Also, in order to ensure that the algorithm is superstabilizing, variables keep their previous assignments during recovery from a fault until a new assignment has been chosen; it is only at this point that their value will change. Thus, in the case of SDPOP, during recovery from a fault, a legitimate system state will be the optimal variable assignment from before the fault.

2.2.5.2 Anytime Properties

Anytime algorithms (Dean and Boddy, 1988) are a class of algorithms intended for time-dependent planning problems (e.g., search and rescue, vehicle monitoring and signal processing). In such planning problems, it is more important to get a good solution quickly, as opposed to an optimal solution slowly. Anytime algorithms were defined on the principle that they will do the best they can in the time available. Hence, an anytime algorithm can be stopped at any time in its computation, and an approximate solution gained. In more detail, anytime algorithms are useful in evolving environments such as ours because they explicitly model the tradeoff between solution quality and solution speed.

To put this in context, imagine a scenario in which ten civilians are in a building that is on fire, and only two firefighters are available to rescue them. If those firefighters used an allocation algorithm that was not anytime in order to decide which civilians to rescue in which order, they might have to wait a minute or more to find out which civilian to rescue first. In this time, the fire could progress further, making access to one or more civilians impossible. At this point, the firefighters face the decision between running the algorithm again and wasting more time, or going on the first allocation, which could include the unreachable civilians. In contrast, if an anytime algorithm were used, the firefighters could obtain an allocation at any time, without having to wait.[9] Similarly, when the fire progresses, the firefighters would not need to wait for a new solution, instead continuing with the previous one, checking the result of the algorithm at regular intervals.

The optimal DCOP algorithms we consider (i.e., DPOP and ADOPT) are not anytime, because they are run once, run to completion, and an optimal solution is obtained at the end. Conversely, the approximate algorithms mentioned earlier (specifically, DaCSA, DALO, and max-sum) are all examples of anytime algorithms, since variable values can be chosen based on the messages received up to when the algorithm is stopped.

2.2.5.3 Competitive Analysis

Competitive analysis, introduced by Sleator and Tarjan (1985), is used to analyse online algorithms. In more detail, online algorithms can process any arbitrary input as it is received (i.e., in real time), and continue to operate afterward. Such online algorithms are often needed in distributed systems (Aspnes, 1998), and/or real-time scheduling applications (Palis, 2004), which are similar to our domain of dynamic task allocation, in which updates to the environment could happen at any time.

[9] Assuming some initial solution has been defined: for example, each firefighter attending to the closest/easiest to save civilian.

Now, the objective of competitive analysis is to discover whether or not an online algorithm is *competitive*. In order to define what competitive means, we must first explain the concept of *competitive ratio*. In more detail, the competitive ratio is a metric measuring the difference between the performance of an online algorithm and an optimal offline algorithm, in an environment undergoing arbitrary changes. In this case, the optimal offline algorithm is given full knowledge of the changes to be made to the environment, whereas the online algorithm is only aware of the changes as they happen. As the competitive ratio tends to 1, the online algorithm tends toward being optimal.

Given this, an online algorithm is said to be competitive if the competitive ratio is bounded. In order to discover the competitive ratio of an algorithm, competitive analysis uses the concept of an adversary: an 'opponent' who carries out the changes on the environment. This adversary will try to make the most difficult changes possible in order to discover the worst case performance of the algorithm being analysed. In our work, the environment itself is the adversary, as it changes over time, adding or removing agents and tasks.

As yet, no competitive analysis has been performed for any of the algorithms we consider. However, since only DPOP has an online extension (SDPOP), and both DPOP and SDPOP are optimal algorithms, the competitive ratio of SDPOP will be 1 when competitive performance is measured in terms of utility.

Now that we have explained our three key concepts (self- and superstabilization, anytime algorithms and competitive analysis), we can attempt to design algorithms with these properties in order to meet our requirements.

2.2.6 Discussion

This section provides a short discussion of our findings on the algorithms we have detailed. A summary of these findings is given in Table 2.1, which shows a comparison of the algorithms we consider, and assesses them against our requirements, set out in Section 1.3.[10]

Requirement	DPOP	ADOPT	DaCSA	DALO	max-sum
1. Scalable	10s	10s	100s	100s	100s
2. Robust	Yes	Yes	Yes	Yes	Yes
3. Efficient	No	No	No	No	No
4. Adaptive	Yes	No	No	No	No
5. Quality	Optimal	Optimal	Approximate	Approximate	Approximate
6. Bounded	—	—	Yes	Yes	Yes

Table 2.1: Comparison of key DCOP solution algorithms, against the requirements laid out in Section 1.3.

[10] When we talk about scalability, we are measuring it in terms of numbers of agents and tasks.

It is clear from the table that the complete algorithms (DPOP and ADOPT) do not satisfy our scalability requirement, as increasing the number of agents exponentially increases the time and bandwidth needed to find a solution. As such, only DaCSA, DALO and max-sum (specifically, BnB MS) satisfy Requirement 1. All algorithms also automatically satisfy Requirement 2 because the algorithms being decentralised was a basic selection criteria. However, the complete algorithms are not strictly decentralised, because removing a key node in either algorithm would mean that the entire algorithm would need to be re-run. To be more specific, DPOP and ADOPT rely on agents being arranged in a DFS tree, which would need to be re-computed if any non-leaf agent were to be removed. In addition, in DALO, all local functions are sent to one leader node, which decides the local assignment. If one of the leader nodes were to be removed from the graph, then the groups would have to re-form. In terms of Requirement 3 (efficiency of communications), whilst all the algorithms use only local communication, none of them specifically aim to be efficient in terms of communication, especially after a change in the graph where it is likely that some messages will not be needed to be sent again, and, as such, none of the algorithms can meet our requirement. Only DPOP has explicitly tackled the issue of adapting to changes in the environment (Requirement 4). In terms of solution quality (Requirement 5), max-sum is on the borderline between complete and approximate algorithms: on tree-structured graphs, it has been proven to find the optimal solution, and on cyclic graphs, the BMS extension to max-sum can be used to find good quality bounded approximate solutions (Requirement 6).

Against this background, we can see that max-sum, BnB MS, and BMS satisfy Requirements 1, 2, 5 and 6 (scalable, robust, good quality solutions, bounded), but need to be improved to scale further, be adaptive in environments that change over time, and be efficient in their use of communication, particularly when the environment changes (Requirements 3 and 4). Hence, these are the algorithms that will form our point of departure.

2.3 Max-sum, BnB Max-sum, GHS and Bounded Max-sum

In this section, we elaborate on the technical details behind the algorithms we have chosen to augment: max-sum, BnB MS, GHS (the maximum spanning tree algorithm used by BMS) and BMS. Now, it should be noted that GHS is not strictly central to BMS, and can be replaced by any other maximum spanning tree algorithm. However, GHS is a distributed, asynchronous algorithm for general, undirected graphs, and is optimal in terms of communication cost, and has a linearithmic run-time.

2.3.1 Formal Definitions

In order to explain max-sum, BnB MS and BMS, we must first provide some formal definitions.[11] As we said in Section 2.2.1, max-sum, BnB MS and BMS rely on the environment being represented as a factor graph, which is a bipartite, undirected graph $\mathcal{FG} = \{\mathcal{N}, \mathcal{E}\}$, where $\mathcal{N}$ is the set of nodes, such that $\mathcal{N} = \mathcal{X} \cup \mathcal{F}$, where $\mathcal{X}$ is a set of variable nodes, and $\mathcal{F}$ is a set of function nodes. In addition, $\mathcal{E}$ is a set of edges, where each edge $e \in \mathcal{E}$ joins exactly one node in $\mathcal{X}$ to exactly one node in $\mathcal{F}$.

In this factor graph, there is a set of variables, $\mathcal{X} = \{x_1, \ldots, x_m\}$ which take values from a set of discrete domains, $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$, and a set of functions, $\mathcal{F} = \{f_1, \ldots, f_n\}$, where $f_j(\mathbf{x}_j)$ denotes the value for a possible assignment of the parameters to function f_j, denoted $\mathbf{x}_j$.

Given this, the aim is to find the state of each variable in $\mathbf{x}$ which maximises the sum of all functions in the environment (known as social welfare):

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \sum_{f_i \in \mathcal{F}} f_i(\mathbf{x}_i) \tag{2.1}$$

2.3.2 Max-sum

We now describe the max-sum algorithm in more detail, using the definitions given in Section 2.3.1. Max-sum relies on two types of messages: those from variable to function, q_{ij}, and from function to variable, r_{ij}. We detail these messages below.

From variable to function:

$$\begin{gathered} q_{i\to j}(x_i) \text{ for all values of } x_i \text{ where:} \\ q_{i\to j}(x_i) = \textstyle\sum_{k\in\mathcal{M}_i\setminus j} r_{k\to i}(x_i) \end{gathered} \tag{2.2}$$

where x_i is a variable with index i, j is the index of function f_j, and $\mathcal{M}_i$ is a vector of function indices, indicating which function nodes are connected to variable node i.

From function to variable:

$$\begin{gathered} r_{j\to i}(x_i) \text{ for all values of } x_i \text{ where:} \\ r_{j\to i}(x_i) = \max_{\mathbf{x}_j\setminus i} \left[f_j(\mathbf{x}_j) + \textstyle\sum_{k\in\mathcal{N}_\mathbf{j}\setminus i} q_{k\to j}(x_k) \right] \end{gathered} \tag{2.3}$$

where, f_j is a function with index j, i denotes the index of variable x_i, $\mathcal{N}_j$ is a vector of variable indexes, indicating which variable nodes are connected to function node j and

[11] These definitions are given for ease of description of max-sum, BnB MS and BMS — we give our own problem description and definitions in Chapter 3.

$\mathbf{x}_j \backslash i \equiv \{x_k : k \in \mathcal{N}_j \setminus i\}$.

In more detail, these messages contain sets of values (one for each possible variable assignment) representing the total utility of the system for each possible value of each variable. At any time during propagation of these messages, an agent can determine which value its variable should take by maximising the sum over all variable state utilities. To do this, max-sum locally calculates its marginal function $z_i(x_i)$ from the messages received at each agent. Formally, this function is defined as:

$$z_i(x_i) = \sum_{j \in \mathcal{N}_i} r_{j \to i}(x_i) \tag{2.4}$$

and hence finds $\arg\max_{x_i} z_i(x_i)$.

It can be seen from the messages that max-sum finds the solution to a global optimisation problem through only local communication and computation. It does not involve any preprocessing phase to determine a root/leader node (as in ADOPT or DPOP), and as such, if any node is removed from the graph, the algorithm can continue to operate. Now, max-sum runs continuously, which means that if any change is made to the factor graph, changing the utility received from some factors, then the factor nodes will recompute and send messages, in order to find a new solution. However, max-sum does not specifically cater for changing environments, so there are no defined behaviours for max-sum after a change in the environment. The simplest response to a change in the environment would be to re-run max-sum after every change; however, this could incur unnecessary communication and computation. In particular, the recalculation and resending of messages is inefficient, specifically, in the maximisations in Equations 2.3 and 2.4. In particular, factors and variables will always recompute and resend solutions, even if the messages they have received do not change values, or the solution. Such recomputations can be expensive, particularly at the factors, since each factor must explore the complete space of possible variable assignments in order to find the best solution to send. This space of assignments is exponential in the number of values for each variable for a given factor. The issue of function computation is addressed by BnB MS, which we detail in the next section — however, the inefficiency caused by changes in the environment has yet to be solved, so we intend to address this in our work.

2.3.3 BnB Max-sum

BnB MS (Stranders et al., 2009) consists of two modifications to max-sum. The first of these is an additional preprocessing step of domain pruning at each variable node, which removes variable states which will never lead to the optimal solution (known as *dominated states*) before max-sum runs. The second modification is the use of an online branch and bound technique used when each function node computes its messages, such

that combinations of states that will not be chosen are pruned from the search space. We give further detail on the modifications in the remainder of this section.

The domain pruning preprocessing step uses localised message passing to identify and remove dominated states — i.e., states that will never lead to the optimal solution. Formally, a state $s' \in \mathcal{D}_i$ (where, as mentioned earlier, $\mathcal{D}_i$ is the domain of variable x_i) is dominated if there exists a state $s^* \in \mathcal{D}_i$ such that:

$$\forall \mathbf{x}_{\_i} \sum_{j \in \mathcal{M}_i} f_j(s', \mathbf{x}_{\_i}) \leq \sum_{j \in \mathcal{M}_i} f_j(s^*, \mathbf{x}_{\_i}) \tag{2.5}$$

where $\mathbf{x}_{\_i}$ denotes an assignment of all of the variables in $\mathcal{X}$ *except* x_i. Thus, we can be certain that a given state is dominated if there exists another state which always produces higher utilities than it.

Algorithm 2.1 Algorithm for computing BnB MS domain pruning message from function f_j to variable x_i.

1: Compute $f_j(x_i)^{ub} \geq \min_{\mathbf{x}_{\_i}} f_j(x_i, \mathbf{x}_{\_i})$
2: Compute $f_j(x_i)^{lb} \leq \max_{\mathbf{x}_{\_i}} f_j(x_i, \mathbf{x}_{\_i})$
3: Send $< f_j(x_i)^{ub}, f_j(x_i)^{lb} >$ to x_i

Domain pruning is decentralised through message passing between function and variable nodes, as shown in Algorithms 2.1 and 2.2. The algorithm begins with each function f_j estimating its upper and lower bounds for $x_i = s$, for all $s \in \mathcal{D}_i$ (lines 1 and 2, Algorithm 2.1), and sending them to x_i (line 3), for each x_i neighbouring f_j. This bound estimation is specific to the shape of the functions, but, in the worst case, the bounds can be computed by maximising/minimising over all $\mathbf{x}$ for each value of x_i.

Algorithm 2.2 Algorithm for computing BnB MS domain pruning message from variable x_i to all neighbour functions $f_j, j \in \mathcal{M}_i$.

1: **if** A new message has been received from all $f_j \in \mathcal{M}_i$ **then**
2: Compute $\bot(x_i) = \sum_{j \in \mathcal{M}_i} f_j(x_i)^{lb}$
3: Compute $\top(x_i) = \sum_{j \in \mathcal{M}_i} f_j(x_i)^{ub}$
4: **while** $\exists s \in \mathcal{D}_i : \top(x_i = s) < \max \bot(x_i)$ **do**
5: $\mathcal{D}_i \leftarrow \mathcal{D}_i \setminus \{s\}$
6: **end while**
7: Send updated domain $\mathcal{D}_i$ to each $f_j, j \in \mathcal{M}_i$
8: **end if**

Once a variable node has received a message from each of its adjacent functions (line 1, Algorithm 2.2), it creates its own upper and lower bound, denoted $\top$ and $\bot$, on each value in its domain, $\mathcal{D}_i$. These bounds are computed by summing together the upper and lower bounds it has received from its neighbours for each value in its domain (lines 2 and 3, Algorithm 2.2). The variable node then uses the $\top$ and $\bot$ values to prune dominated states whose upper bounds are lower than the biggest computed lower bound (lines 4 and 5, Algorithm 2.2). Finally, if the variable's domain has changed, then the

updated domain is sent to neighbouring function nodes, and the domain pruning process begins again.

Eventually, domain pruning will converge to a state where variable domains are no longer updated — at which point, max-sum message passing begins at the variable and function nodes, and is, for the most part, as described in Section 2.3.2. The main change here is that BnB MS speeds up the computation of the messages sent from function nodes to variable nodes (Equation 2.3), which have an expensive maximisation over all joint states of the variables adjacent to a function node in the factor graph. This speedup is obtained through the use of branch-and-bound on search trees.

In more detail, as given previously in Equation 2.3, when a function f_j computes its message to send to a variable x_i in max-sum, it does the following:

$$\begin{gathered} r_{j\to i}(x_i) \text{ for all values of } x_i \text{ where:} \\ r_{j\to i}(x_i) = \max_{\mathbf{x}_j \setminus i} \left[f_j(\mathbf{x}_j) + \sum_{k \in \mathcal{N}_\mathbf{j} \setminus i} q_{k\to j}(x_k) \right] \end{gathered} \tag{2.6}$$

where, as before, $\mathcal{N}_j$ is a vector of variable indexes, indicating which variable nodes are connected to function node j and $\mathbf{x}_j \setminus i \equiv \{x_k : k \in \mathcal{N}_j \setminus i\}$. This maximisation is a computational bottleneck, because the function node must enumerate all valid states and choose the one with the maximum utility. This is not as simple as using the maximum value of f_j, because the maximisation includes messages received from variables. Hence, to reduce this state space, a search tree is built, where each node is a partial joint state $\hat{\mathbf{x}}$ (where some variable states are not yet set), and the leaf nodes are full joint states (where the values of all variables are set). Branch-and-bound can then be applied to this search tree, by computing the maximum value that can be obtained from each branch, and pruning dominated branches.

To do this, first, re-formulate the function message:

$$\begin{gathered} r_{j\to i}(x_i) \text{ for all values of } x_i \text{ where:} \\ r_{j\to i}(x_i) = \max_{\mathbf{x}_j \setminus i} \tilde{r}_{j\to i}(\mathbf{x}_j) \end{gathered} \tag{2.7}$$

where:

$$\tilde{r}_{j\to i}(\mathbf{x}_j) = f_j(\mathbf{x}_j) + \sum_{k \in \mathbf{adj(f_j)} \setminus i} q_{k\to j}(x_k) \tag{2.8}$$

Hence, in order to compute an upper bound, the maximum $\tilde{r}_{j\to i}(\mathbf{x}_j)$ must be found for all $\mathbf{x}_j \setminus i$, and to compute a lower bound, the minimum must be found.

Given this, the branch-and-bound algorithm works as follows: to begin, a search tree is constructed, rooted at the partial joint state $\hat{\mathbf{x}}_r = \{-, \ldots -, x_i, -, \ldots, -\}$. Then, the

first unassigned variable in $\hat{\mathbf{x}}_r$ (say, x_1), is chosen, and a branch of the tree is created for each possible value of that variable. Next, upper and lower bounds are estimated for the value of $\tilde{r}_{j\to i}(\hat{\mathbf{x}})$ for each of those branches. The bounds for each branch are then assessed, and branches where the upper and lower bounds are lower than the smallest lower bound at that level of the tree are pruned. Following this, all branches which haven't been pruned are expanded, and the algorithm continues until the leaves of the search tree are reached, where the leaves are fully completed joint states (i.e., one which contains only fixed variable values). $\tilde{r}_{j\to i}$ is then computed for each of these joint states so that the maximum value can be chosen with much less computation needed.

Now, as there are limited results as to the convergence of GDL algorithms on cyclic graphs (Wainwright et al., 2004), it cannot be guaranteed that max-sum and BnB MS will converge to a solution on a cyclic graph. Thus, the BMS algorithm combats this issue of convergence on cyclic graphs by finding a maximum spanning tree of the graph using the GHS algorithm (Gallager et al., 1983), and runs max-sum on that spanning tree. This allows BMS to produce quality guarantees on the solutions it gives.

2.3.4 GHS

In order to explain BMS in more detail, we first explain GHS, which is the maximum spanning tree (MST) algorithm employed by BMS in phase 1 of its operation. As explained earlier, GHS is optimal in terms of communication cost $O(n \log n + E)$ and has a linearithmic run time $O(n \log n)$ where n is the number of nodes in the factor graph, and E is the number of edges. We give the complete pseudocode for GHS in Appendix A, and refer to it throughout this section.

The GHS algorithm (Gallager et al., 1983) is a distributed technique to find the MST of any given weighted undirected graph, using only local communication and low computation at each node. It should be noted that, although GHS is designed to find a *minimum* spanning tree, it can simply find a maximum spanning tree by negating edge weights. In addition to this, in graphs where edge weights are not distinct (and thus, there could be multiple MSTs), GHS will only successfully compute an MST if the edge weights are made distinct. By this, we mean that the weight of an edge is to be treated as a tuple $(w, \langle i, j\rangle)$, where w is the edge weight and i and j are the node IDs at either end of the edge. Then, comparisons between edge weights are done based on their weight, and the node IDs at either end.

Now, the GHS algorithm takes as input a weighted, undirected graph, and outputs the minimum spanning tree of that graph. Given this graph, the GHS algorithm operates, as most MST algorithms do, by finding a spanning forest of rooted trees, beginning with individual nodes, and gradually joining them together on their lowest weight edges. These rooted trees are referred to as *fragments* of the MST in GHS, as they are subtrees

of the eventual MST. Figure 2.2 shows an example of how fragments can join, and introduces the concept of *levels* of fragments. As can be seen from the Figure, when two fragments (F and F′) at the same level (1), join to form a new fragment, F″, the level of the new fragment is incremented by 1.

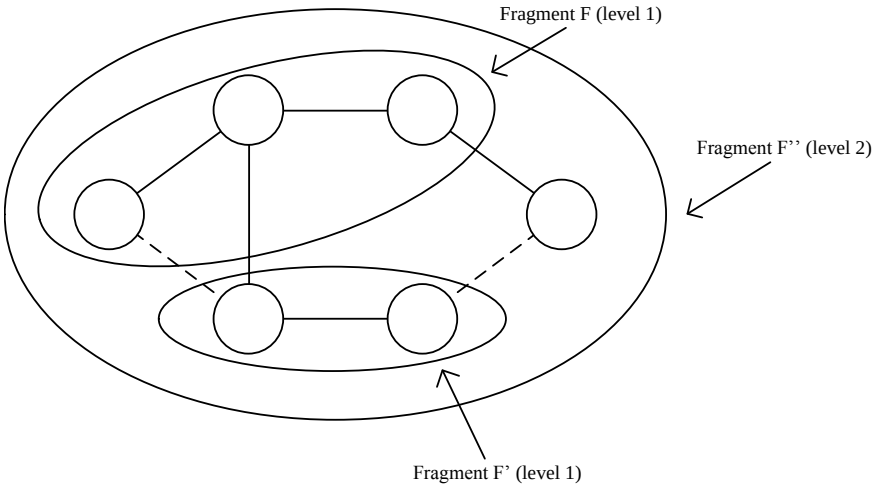


FIGURE 2.2: Illustration of levels and fragments in the GHS algorithm. Adapted from (Gallager et al., 1983).

In more detail, the GHS algorithm stores a number of values at each node in order to find the MST. These values are:

- SN: the node's current state. Either *Sleeping* (if it hasn't received any messages yet), $Find$ (if looking for a minimum weight outgoing edge), or $Found$ (if it has found a minimum weight outgoing edge, or does not have one).
- SE: a map of states for every edge incident on the node. Edge states can be either $Basic$ (if not categorised yet), $Branch$ (if decided to be a spanning tree branch), or $Rejected$ (if decided not to be in the spanning tree.)
- FN: the ID of the fragment the node is currently in. This takes the form of an edge weight.
- LN: the level of the fragment the node is currently in.
- $bestEdge$, $bestWt$: the minimum outgoing edge, and its weight, as found by the node.
- $testEdge$: the edge currently being tested to see if it is the minimum outgoing edge.
- $inBranch$: the edge directed up to the root of the tree.
- $findCount$: the number of edges we are waiting for a response on.

Given the above parameters, GHS uses a number of messages. These are:

- $Connect(fragmentID, level)$: sent over an identified minimum weight outgoing edge in order to combine two fragments.
- $Initiate(fragmentID, level)$: used in order to broadcast a node's new $fragmentID$ and level, and start minimum weight outgoing edge search in the node's fragment.
- $Test(fragmentID, level)$: used to find out if an edge e leads to another fragment (i.e., a node with a different $fragmentID$).
- *Accept*: used as a response to a $Test$ message, to indicate the edge leads to a node in a different fragment.
- *Reject*: used as a response to a $Test$ message, to indicate the edge leads to a node within the current fragment.
- $Report(weight)$: used to inform the node's parent of the minimum weight outgoing edge of the subtree rooted at the node.
- $Change - root$: sent from the root of the current fragment to the identified minimum weight outgoing edge. This informs the node with the minimum weight outgoing edge to connect through it.

Now, the algorithm begins with all nodes in the state *Sleeping*. The `wakeup` procedure (Algorithm A.1) will be executed at one node, initialising its variables. The node will then choose its minimum weight outgoing edge and send a *Connect* message along that edge in order to expand the current fragment. The receiving node will call the procedure given in Algorithm A.2, wake itself up (and in doing so, send a *Connect* message to its own minimum weight outgoing edge), and decide whether to connect to the original node or not. This decision is informed by the current level of the two nodes. If one is higher than the other, then a connection can be made, and an $Initialise$ message is sent. If not, then the node will postpone processing the message until this is true. When a node receives an $Initialise$ message (Algorithm A.3), it is informed of its new fragmentID and level, and must start testing each of its remaining non-$Branch$ edges in order to find which connect to nodes with different $fragmentID$ values. This is done by sending $Test$ messages. When a node receives a $Test$ message (processed in Algorithm A.5) it will reply with *Accept* (processed in Algorithm A.6) if its $fragmentID$ value is not equal to that which it was sent. If the $fragmentID$ values are equal, then the node replies with *Reject* (processed in Algorithm A.7).

When a node has tested all of its edges, and ascertained which lead to other fragments, it runs the `report` procedure (Algorithm A.8) in order to calculate its minimum weight outgoing edge, and send it to its parent node in a *Report* message (processed

in Algorithm A.9). These messages then propagate up to the root of the current fragment, at which point the minimum weight outgoing edge of the fragment is known. A *Change − root* message (processed in Algorithm A.11) is, thus, sent toward the source of the minimum weight outgoing edge. When the source node is reached, it runs the `change-root` procedure (Algorithm A.10) in order to merge with the fragment at the end of the edge.

This process then continues until all nodes in the graph are members of the same fragment, and no minimum weight outgoing edge exists. At this point, there exist two nodes considered as the root — one node either side of the final connection made. Thus, whichever of these nodes is a factor must send a COMPLETE message to its children informing them that the algorithm has completed. Nodes receiving this message mark the sender as their parent, and send the message on to their children. When the leaves receive this message, the preprocessing has completed.

Next, we show how BMS uses the MST computed by GHS to provide bounded approximate solutions.

2.3.5 Bounded Max-sum

In the remainder of this section, we describe BMS (Rogers et al., 2011), with respect to the formal definitions we gave in Section 2.3.1. Now, the first step of BMS is to weight all of the edges in the factor graph, in order to enable the algorithm to detect which edges contribute most to the overall system utility. Each dependency link e_{ij} in a factor graph is weighted as below:

$$w_{ij} = \max_{\mathbf{x}_i \setminus j} \left[\max_{x_j} f_i(\mathbf{x}_i) - \min_{x_j} f_i(\mathbf{x}_i) \right] \tag{2.9}$$

This weight represents the maximum impact that variable x_i can have over function F_j. Thus, by not including link e_{ij} in the spanning tree, we say that the distance between the solution and the optimal is at most w_{ij}.

Next, the GHS algorithm, as described above in Section 2.3.4, is run over the graph in order to find an MST. It is preferable to formulate the graph as a MST in order to try and obtain a better approximation at the optimal solution, thus improving the solution quality. In more detail, when constraints are binary, the approximation ratio (the distance between the solution given and the optimal) is minimised through operating the algorithm over a maximum spanning tree — i.e., the bound is most accurate where constraints are binary (Farinelli et al., 2009).

After the preprocessing phase, each factor node is aware of which of its edges are in the MST and which are not. The edges not in the MST are stored in the set $\mathbf{x}_i^c$. Conversely,

the set of edges that have not been removed is denoted $\mathbf{x}_i^t$, and thus, it follows that $\mathbf{x}_i = \mathbf{x}_i^c \cup \mathbf{x}_i^t$.

In the next step of BMS, the max-sum algorithm is run on the MST, beginning at the leaves, propagating up the tree, with each node waiting to receive messages from all of its children before calculating and sending its next message. These messages then propagate back down the graph from the root to the leaves in a similar manner, at which point the algorithm converges. Functions with removed dependencies are evaluated by minimising over the removed dependencies, as follows:

$$\tilde{\mathbf{x}} = \arg\max_{\mathbf{x}} \sum_i \min_{\mathbf{x}_i^c} F_i(\mathbf{x}_i) \tag{2.10}$$

where $\tilde{\mathbf{x}}$ denotes the approximate solution: the solution gained through applying max-sum to a spanning tree of the original graph. Following this step, each node is aware of $\tilde{\mathbf{x}}$, and $\tilde{V}^m = \sum_i \min_{\mathbf{x}_i^c} F_i(\tilde{\mathbf{x}}_i)$, which will eventually be used to calculate the approximation ratio of the graph, $\rho(FG)$

Once the leaves have received the max-sum messages, they can compose new WSUM and SOLUTION messages. If the leaf node is a function, WSUM is the sum of the weights of its removed dependencies, and SOLUTION is $F_i(\tilde{\mathbf{x}}_i)$. The sum of removed weights, W, is defined as follows:

$$W = \sum_{e_{ij} \in C} w_{ij} \tag{2.11}$$

where C is the set of links removed from the factor graph. Now, if the leaf is a variable, the WSUM and SOLUTION messages are empty. When a node receives WSUM and SOLUTION from all its children, it can process them according to whether it is a function node or a variable node. If the node is a variable node, these messages are the sum of messages from its children. If the node is a function node, the messages are the sum of messages from its children, plus the weights of its own removed edges, and $F_i(\tilde{\mathbf{x}}_i)$. Once these messages reach the root, the root propagates them back down, so every node is aware of the total weight removed, W, and the solution value, $\tilde{V} = \sum_i F_i(\tilde{\mathbf{x}}_i)$.

Now the agents have enough information available to them, the approximation ratio can be calculated at each node, as follows:

$$\rho(FG) = 1 + (\tilde{V}^m + W - \tilde{V})/\tilde{V} \tag{2.12}$$

When every node is aware of this approximation ratio, the algorithm is complete.

Now, whilst max-sum, BnB MS, GHS and BMS are useful for our purpose, in order to satisfy our requirements laid out in Section 1.3, we must extend them in order to be faster and cope with an environment which changes over time. In more detail, max-sum, BnB MS and BMS have no specific protocol for when a change occurs in the environment

(and so, the factor graph representing the environment), and so should be run again from scratch in order to find a new allocation of variables to values. This can incur communication and computation which is not necessary. In particular, if a variable is assigned to the same value before and after a change in the graph, then all messages sent to and from that variable are unnecessary (as they do not change the allocation), or if a function's message values are not effected by the change in the graph, then any recomputation and communication here is unnecessary, too. Hence, special provision must be made for the algorithms to be used in environments that change over time, in order to avoid unnecessary overheads being incurred in environments that change rapidly.

In addition to the adaptivity and communication efficiency issues presented by the use of max-sum, BnB MS and BMS, some particular factor graph formulations of task allocation problems can give rise to a problem of deciding which agents should perform the computation and communication of function nodes, where each function node relates to a task in the environment — if more than one agent could perform the computation of a given function node, the problem here is how to decide which agent could compute the function with the least impact on the overall run-time of the algorithm. This is an important problem to solve in order to make these algorithms applicable in the real world, as, obviously, a rescue task (for example, a fire to be extinguished, a civilian to be rescued, or a target to be tracked by a mobile sensor) is not able to perform any communication or communication by itself. The computation and communication representing each rescue task must therefore be distributed amongst agents. Now, this distribution must be found by the agents whilst taking into account the fact that the rescue environment can and will change over time. Therefore, in the next section, we go into more detail on this problem of *computation distribution*, in order to allow our own mechanism to be applied in real life rescue environments, which change over time.

2.4 Distribution of Computation

In more detail, the computation distribution problem is found in a factor graph formulation which is particularly expressive and intuitive for application to task allocation problems, in which tasks are represented as function nodes. However, existing work is yet to specifically tackle the problem of which agents should perform the computation and communication of function nodes (given that the tasks themselves cannot perform this). This introduces a second optimisation problem to the initial task allocation problem: a problem of allocating the computation of a solution to a DCOP. Specifically, the problem is to place function nodes onto agents' computational devices in such a way that time taken to compute a solution to the DCOP is minimised. In considering a problem with such a concept of time, we allow this formulation to be used in other cases, too — for example, as battery power used is a function of computation and communica-

tion time (i.e., the more computation/communication an agent performs, the more their battery is depleted), this formulation could also be applied to mobile sensor network environments, by converting sensor battery expenditure estimates into computation and communication time estimates. Another example of the generality of our concept of time can be found in scenarios where communication between agents is unreliable — here, the quality of the communication link is directly correlated to communication and computation time.

A similar problem to ours has been tackled recently by Stefanovitch et al. (2010): specifically, the work tackles the problem of building junction trees to schedule computation of GDL algorithms across agents. In more detail, Stefanovitch et al. (2010) schedule computation and communication of a set of cliques of variables, that are independent of the underlying problem, and subject to communication and computation constraints. In so doing, computation representing both agents and tasks in the environment is scheduled across the agents. This is in contrast to our approach, where we need only schedule the computation of factors across the agents, since the variable nodes are already allocated to the agent they represent. In addition, the approach we take is to generalise the computation distribution problem to a well known operations research problem (known as $R||C_{\max}$, explained shortly), thus simultaneously solving our own computation distribution problem and providing a meaningful contribution to the operations research community.

In investigating how to solve this computation distribution problem, we found that this problem has been studied by the operations research community. We elaborate on this problem in the remainder of this section.

2.4.1 Scheduling on Unrelated Parallel Machines ($R||C_{\max}$)

This work is concerned with finding solutions to a common scheduling problem known as the *scheduling on unrelated parallel machines* problem. This problem is denoted by Graham et al. (1979) as $R||C_{\max}$, where R denotes unrelated parallel machines, and $C_{\max}$ denotes that we are interested in the maximum completion time, which is called the *makespan*. $R||C_{\max}$ is formally defined as:

Definition 2.1. Given a set T of tasks, a set P of unrelated processors, and for each $j \in T$ and $i \in P$, $p_{ij} \in \mathbb{Z}^+$, the time taken to process task j on processor i, the problem is to schedule the tasks on the processors so as to minimise the makespan.

$R||C_{\max}$ requires the scheduling of a number of tasks on a number of unrelated processors, and assumes that processors are not identical, so that processing times for tasks can differ between them. Next, we describe current algorithms to solve $R||C_{\max}$, and why they are not suitable for our work.

2.4.2 Algorithms for $R||C_{\max}$

Now, as $R||C_{\max}$ has been shown to be NP-hard (Wotzlaw, 2006), exact solution algorithms such as those reviewed by Wotzlaw (specifically, using cutting planes and branch-and-price) generally do not scale well. For example, Horowitz and Sahni give an exact algorithm for $R||C_{\max}$, with a time complexity exponential in the amount of tasks. Their algorithm uses a branch-and-bound dynamic programming approach which proceeds (for 2 processors) by creating a set of 3-tuples, one for each task, containing the finish time on processors 1 and 2, and an encoded bit string whose ones indicate tasks allocated to processor 1. From all of these 3-tuples, the one with the smallest maximum finish time of processors 1 and 2 is selected. The bit string for this 3-tuple is then used to work out which tasks go on which processor. Horowitz et al. show how to generalise the algorithm to $|P|$ processors, by adding an entry in each tuple for each processor (Horowitz and Sahni, 1976). Now, the worst-case computation time for this algorithm is $\mathrm{O}(\min\{|T|C^G_{\max}, |P|^{|T|}\})$ where $C^G_{\max}$ is a greedy makespan estimate, computed as the finish time of the schedule obtained by assigning each task to the processor on which its execution time is minimal. Scheduling each task on the processor with the minimal execution time is unlikely to produce the optimal solution, so the worst-case run-time for the algorithm given in (Horowitz and Sahni, 1976) is exponential.

Thus, in practice, approximation algorithms (Horowitz and Sahni, 1976; Ibarra and Kim, 1977; Lenstra et al., 1990) are far better suited to finding solutions to $R||C_{\max}$, since they generally find their solutions in less time than optimal algorithms. In more detail, Horowitz and Sahni presented an ϵ-approximate version of their exact algorithm described above, which has a worst-case time complexity of $\mathrm{O}(10^{2l}|T|)$, where l is the smallest integer for which $10^{-l} \leq \epsilon$. The heuristics presented by (Ibarra and Kim, 1977) have a worst-case time complexity of $\mathrm{O}(|T|)$, $\mathrm{O}(|T| \log |T|)$ and $\mathrm{O}(|T|^2)$, which are far more preferable to the time complexity of exact algorithms given above. Despite this, (Ibarra and Kim, 1977) have found an approximate algorithm which can attain a best-case approximation ratio of $\frac{1+\sqrt{5}}{2}$, but this is only for two processors,[12] and thus not applicable to our scenario. Further work in this area (Lenstra et al., 1990) found a 2-approximation algorithm which runs in time bounded by a polynomial in the size of the input. However, the approximation quality of this has since been beaten, for example by Kumar et al. (2009), who present a better-than-2 approximation algorithm.

Unfortunately, despite the clear existence of good centralised approximation algorithms for $R||C_{\max}$, our aim is to find a decentralised solution which takes advantage of sparsity in the environment (i.e., where agents can only compute and communicate for a subset of the tasks) in order to scale well (see Requirement 1, Section 1.3), and is therefore more robust (see Requirement 2). However, there exist no decentralised algorithms for $R||C_{\max}$ as yet. Therefore, our algorithm will mark the first exploration into decen-

[12] The writers were unable to extend it to the case of $m \geq 3$.

tralised mechanisms to solve $R||C_{\max}$, and also the first work to use multi-agent systems to solve the problem.

2.5 Summary

In this chapter, we have provided a review of literature relevant to dynamic task allocation. We began with a review of the four main classes of mechanisms for task allocation; specifically, game theory, decision theory, auctions and DCOPs. From this review, we drew the conclusion that applying a DCOP formulation to our problem would best help us to meet the requirements discussed in Section 1.3, because DCOP formulations lend themselves to inherently distributed problems and cope well under changing environments.

Given this, we compared the main available solution algorithms for DCOPs: namely, DPOP, ADOPT, DaCSA, DALO, and max-sum/BnB MS/BMS. We have chosen max-sum, BnB MS, and BMS as starting points for our work, because they have been shown to address four of the requirements laid out in Section 1.3. In more detail, the max-sum algorithm and its variants satisfy a number of requirements:

1. It provides a decentralised, robust (Requirement 2) DCOP solution method, using localised communication and computation to take advantage of the sparsity of interactions in order to compute the most beneficial state each variable can take.

2. Computation of variable state takes place at the agent which owns the variable, thus allowing the solution to scale to hundreds of agents, as required (Requirement 1). This is facilitated by the use of BnB MS, which reduces the computation done at agents by shrinking the domains of the variables owned by agents.

3. The variable computation described above also means that there will be no single point of failure for the system (Requirement 2).

4. BMS is able to produce quality guarantees for approximate solutions (Requirement 6), and has been shown to produce good quality solutions (Requirement 5).

5. Max-sum, BnB MS and BMS can be applied to environments that change over time (Requirement 4), but do not specifically cater for changes in the environment. Thus, recomputing solutions from scratch would be the obvious response to changes in the environment, which is likely to incur extraneous communication and computation each time the environment is changed.

Nonetheless, the max-sum algorithm does not yet meet Requirements 3 and 4 — that is, efficiency in communication and adaptiveness to changes in the environment. As discussed in Section 2.3, the max-sum algorithm does not have any defined procedure

to adapt to a changing environment. Therefore, we identified three key properties of dynamic algorithms so that we can evaluate how our work meets Requirements 3 and 4: specifically, self- and superstabilization, anytime algorithms, and competitive analysis.

Finally, we identified a problem that can arise from our factor graph formulation, which we call *computation distribution*. Specifically, the computation distribution problem arises where tasks are treated as function nodes, and agents as variable nodes. In this case, tasks are obviously unable to perform max-sum communication and computation themselves, so an agent to perform each task must be found. This is a particularly difficult problem when the number of tasks exceeds the number of agents, since if the computation and communication for all tasks is assigned to a few agents, this will cause max-sum to have a large computation time, since agents can be overloaded. Thus, an algorithm needs to be found to allocate this computation such that the overall max-sum runtime (which is equivalent to the *makespan* concept introduced earlier) is minimised. We identified how this problem is analogous to a common operations research problem known as scheduling on unrelated parallel machines, or $R||C_{\max}$, and showed how there currently exist no decentralised algorithms for $R||C_{\max}$. Thus, existing approaches cannot be used for our work, since they violate Requirements 1, 2 and 4 (scalability, robustness, and adaptiveness).

Hence, in the remaining chapters, we address the above-mentioned issues in the following ways:

- In Chapter 3, we show how dynamic task allocation can be formulated as a factor graph. This, then, allows us to highlight where the computation distribution problem comes from, and present it formally.

- In Chapter 4, we present three algorithms for dynamic task allocation: FMS, BnB FMS and BFMS. In more detail, FMS is our first, superstabilizing and anytime, extension to the max-sum algorithm in order to reduce the communication and computation incurred by changes to the graph. BnB FMS specifically targets the scalability of FMS, and consists of two interleaved sub-algorithms. The first attempts to reduce the number of tasks that an individual agent considers, while the second reduces the number of coalitions that need to be evaluated to find the optimal group of agents (coalition) for a task. Finally, BFMS is anytime and superstabilizing, and provides a bounded approximate solution to dynamic task allocation problems. This is achieved by eliminating dependencies in the constraint functions, according to how much impact they have on the overall solution value. In more detail, we propose iGHS (Section 4.4.1), which computes a MST on subsections of the constraint graph, in order to reduce communication and computation overheads. We then empirically evaluate our algorithms in Section 4.5, to prove that, in combination, they address all of our requirements.

- In Chapter 5, we present our algorithm for computation distribution, which we call ST-DTDA. In more detail, ST-DTDA uses a heuristic to find a ST of the graphical representation of the environment, and runs a GDL algorithm called the min-max algorithm over the ST representation of the problem, to generate approximate solutions with per-instance bounds, through decentralised message passing between agents. We then empirically evaluate three different heuristics for ST-DTDA and prove that they address our requirements.

Chapter 3

Modelling Agent Coordination using Max-sum

In Chapter 2, we argued that the max-sum algorithm has a number of properties that make it very appropriate for coordinating agents for dynamic task allocation. Hence, in this chapter, we formalise the dynamic task allocation problem in order to apply max-sum and develop our coordination mechanisms, as outlined in our aims and objectives (Section 1.3).

In more detail, we first present a flooding scenario to highlight the issues faced by rescuers in a real-world dynamic task allocation problem. Next, we formalise the dynamic task allocation problem as a factor graph, so that max-sum can be applied to it. We then explain how the computation distribution problem described in Section 2.4 can arise from this factor graph formulation. Hence, we then provide a formulation of the computation distribution problem, which is known in operations research as $R||C_{\max}$, as a junction graph, so that another GDL algorithm called min-max can be applied to it.

3.1 Scenario

In June and July 2007, some of the most serious inland flooding since 1947 hit much of the UK, in areas including Yorkshire, the Midlands, Worcestershire, Oxfordshire and Wiltshire. Gloucestershire was the worst hit, with the fire and rescue service alone attending 1,800 incidents in just 18 hours (McKie, 2007) — nearly a quarter of the usual amount of calls received in a year. Hence, the Fire Brigades Union in the UK described the rescue effort for the floods as the biggest in peacetime Britain (Pitt, 2008). Our scenario is loosely based on the events in a small village in Gloucestershire. The reader is referred to Appendix B for more details. In what follows, we elaborate on the circumstances that arise in our scenario:

- The scenario shows how the flood makes getting to the most flooded places difficult for land vehicles. It also shows the *urgency* (in terms of hours and days) with which action needs to be taken by rescue agencies (Requirements 4 and 5): without fresh water to drink, lives will be lost.

- In addition, the scenario shows that common lines of communication can be *affected* by flooding. This makes it difficult for civilians to be updated on the progression of the flood and rescue effort, and also for the civilians to inform the rescuers that they need to be saved. The rescue agencies themselves often have to rely on only *short-range* communication, such as walkie-talkies, which are likely to be subject to *interference* from the use of other radio devices. This highlights the need for an algorithm that uses short-range communication effectively in order to introduce sparsity and therefore be robust to interference and loss of communication (Requirement 3).

- The scenario also shows that often, the number and potential capacity of rescue vehicles is lower than the number of civilians that need to be rescued. This, combined with the dynamism in the environment (more tasks will appear over time, rescue agencies may become incapacitated), means that some rescue agencies will have to *prioritise* some tasks over others in order to rescue the most vulnerable civilians, such as the elderly or disabled, quickly. This highlights the need for an algorithm that can find high quality solutions efficiently, while providing support for reallocation of tasks (Requirements 4, 5 and 6).

- Finally, the scenario shows the need for *different* rescue agencies to coordinate and collaborate in order to save more lives (Requirements 1 and 2). If agencies were to work together, the combined manpower could provide a variety of useful capabilities. For example, as mentioned in the scenario, equipping a helicopter with medical personnel could mean that only the most critically ill civilians would need to be airlifted to hospital, whereas others could be treated on the ground. Another example would be for the fire brigade to help uncover rubble from a collapsed building, so that ambulance personnel could rescue people trapped inside. A final example, as seen in the scenario, is beer companies using their tankers to transport water to towns and villages to aid other rescue agencies in the transport of water.

Before going on to specify a solution that aims to address the above issues in Chapter 4, we next formalise the dynamic task allocation problem, with reference to this scenario.

3.2 Problem Description

We denote the set of agents (rescuers in our scenario) as $\mathcal{A} = \{a_1, \ldots, a_{|A|}\}$, who are given a number of tasks (civilians to rescue in our scenario), $\mathcal{T} = \{t_1, \ldots, t_{|T|}\}$. An agent

$a_i \in \mathcal{A}$ can perform any one task in $\mathcal{T}_i \subseteq \mathcal{T}$, the set of tasks a_i can perform. Similarly, we denote the set of agents that can perform task t_j as $\mathcal{A}_j$.

Now, it is likely that the number of tasks will exceed the number of available agents. Agents are therefore encouraged to work together on the tasks, as the utility gained at a task can differ depending on the number of agents assigned to it, and also the specific set of agents assigned to it. We refer to a group of agents collaborating together as a coalition $C \in 2^{\mathcal{A}}$. Now, the utility gained (in our scenario, this corresponds to whether or not the civilian is rescued) at task t_j when it is performed by coalition (which means a group of agents) $C_j \in 2^{\mathcal{A}_j}$ is denoted as $V(C_j, t_j) \in \mathbb{R}^+$. This value is not necessarily additive in the capabilities of the agents in C_j (i.e. two agents could work particularly well/badly together, thus gaining more/less utility than simply summing the values of the agents doing the task alone). Since task t_j cannot be performed with no agents assigned to it, $V(\emptyset, t_j) = 0, \forall t_j \in \mathcal{T}$.

Moreover, we define the *contribution* of a_i to t_j as $\delta_j(a_i, C_j)$, if coalition C_j performs t_j, where $\delta_j(a_i, C_j) = V(C_j, t_j) - V(C_j \setminus \{a_i\}, t_j)$. Given this, and assuming that task utilities are independent (this allows our formulation to be completely general and be used to model both independent and non-independent task utilities), our objective is to find the coalition structure $\mathcal{S}^* = \{C_1^*, \ldots, C_{|\mathcal{T}|}^*\}$ which maximises the global utility:

$$\mathcal{S}^* = \arg\max_{\mathcal{S}} \sum_{C_j \in \mathcal{S}} V(C_j, t_j) \tag{3.1}$$

where $\mathcal{S}$ denotes the set of all coalition structures, and subject to the constraint that no two coalitions overlap (i.e., no agent does more than one task at a time):

$$C_j \cap C_k = \emptyset, \forall C_j, C_k \in \mathcal{S}, j \neq k \tag{3.2}$$

This assumption of non-overlapping coalitions is important, since if it were relaxed then that would mean that an agent could be in two places at once, doing two tasks at once, which is illogical.

The state space that Equation 3.1 covers is potentially very large, i.e. $\prod_{i=1}^{|\mathcal{A}|} |\mathcal{T}_i|$. However we can exploit the fact that $\mathcal{T}_i \neq \mathcal{T}$, i.e., the problem is factorisable due to the fact that it is unlikely that every agent will be able to perform every task, and ensure that there is no dependence on a central entity by decentralising computation. To this end, we now formulate our problem as a DCOP in order to solve it using max-sum.

3.3 DCOP Formulation

A DCOP is defined as a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F} \rangle$, where $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ is a set of agents, $\mathcal{X} = \{x_1, \ldots, x_{|\mathcal{X}|}\}$ is a set of variables, $\mathcal{D} = \{D_1, \ldots, D_{|\mathcal{D}|}\}$ is a set of discrete and finite

variable domains, and $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$ is a set of functions describing constraints among the variables. In more detail, each variable $x \in \mathcal{X}$ is owned by precisely one agent $a \in \mathcal{A}$. However, each agent can potentially own more than one variable. Now, each variable x_i can take a value in the domain D_i, and the functions in $\mathcal{F}$ act to restrict these domains.

In yet more detail, each function $f_i : D_{i_1} \times \cdots \times D_{i_{r_i}} \to \mathbb{R}$ depends on a set of variables $\mathbf{x}_i \subseteq \mathcal{X}$, where $r_i = |\mathbf{x}_i|$ is the arity of the function. Thus, each function assigns a real value to each possible assignment of the variables it depends on.

Now, there are a number of ways in which we could formulate our problem as a DCOP, which can have an impact on communication and computational load, as shown by Maheswaran et al. (2004). We choose to assign each agent $a \in \mathcal{A}$ a variable, $x_i \in \mathcal{X}$ representing the task the agent is currently allocated to, as opposed to creating a variable for each task (which would take a value in $2^{\mathcal{A}}$), or for each coalition.

In our formulation, variable domains consist of the tasks which agents can perform. More formally, if x_i is the variable owned by agent a_i, then the domain of x_i is $D_i = \{t_k | t_k \in \mathcal{T}_i\}$.

Each function represents the utility of each task, taking into account all variables whose domains contain that task. Formally speaking, given $x_i \in \mathcal{X}$ and $t_j \in \mathcal{T}_i$, we say that x_i is a variable to t_j's function f_j if and only if $t_j \in D_i$. We denote the set of variables in the domain of function f_j domain as $\mathcal{X}_j \subseteq \mathcal{X}$. As such, we have $|\mathcal{A}|$ variables, one for each agent, and $|\mathcal{T}|$ functions, one for each task.

Thus, each function, $f_j(\mathbf{x}_j) \in \mathcal{F}$ represents the task utility function $V(C_j, t_j)$, where $\mathbf{x}_j$ represents an assignment of the variables in $\mathcal{X}_j$: for example, $\{(x_1 = t_1), (x_2 = t_2), (x_3 = t_1)\}$. Thus, $(x_i = t_j) \in \mathbf{x}_j$ is equivalent to $a_i \in C_j$, and so, the objective function of our DCOP, given below in Equation 3.3 is equivalent to Equation 3.1.

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \sum_{f_j \in \mathcal{F}} f_j(\mathbf{x}_j) \tag{3.3}$$

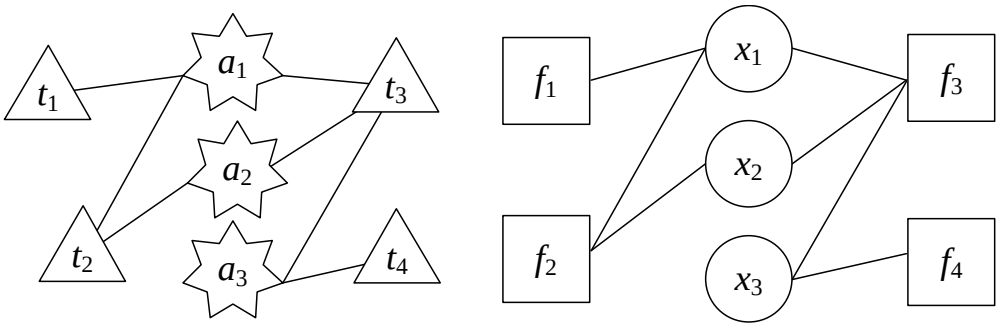
Now that we have formulated our problem as a DCOP, we can formulate it as a factor graph in order to apply the max-sum algorithm to find solutions.

3.4 Factor Graph Formulation

Recall from Section 2.2.1 that a factor graph is a bipartite, undirected graph which provides an intuitive representation of the structure of the factorisation of a utility function. A factor graph consists of two types of node: function nodes (factors) for each local function and variable nodes to represent the variables involved in those functions.

More formally, a factor graph can be represented by a tuple $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$, in which $\mathcal{N} = \mathcal{X} \cup \mathcal{F}$ is the set of nodes, consisting of the set of function nodes $\mathcal{F}$ and the set of variable nodes $\mathcal{X}$, and $\mathcal{E}$ is a set of edges connecting function nodes to variable nodes. There will be an edge connecting function f_j to variable x_i if and only if x_i is a parameter to f_j (Kschischang et al., 2001).

In accordance with our DCOP formulation, given in Section 3.3, there is a variable node for each agent, and a function node for each task in the environment. To illustrate this, Figure 3.1(a) shows an example scenario which we base our formulation on, and Figure 3.1(b) shows our particular formulation, which we elaborate on below.



(a) Example scenario. (b) Factor graph of scenario.

FIGURE 3.1: An example scenario, and factor graph of the scenario, containing 3 rescue agents (stars) and 4 rescue tasks (triangles). In the factor graph, variables are denoted by circles, and factors as squares

This formulation, although perhaps not the most intuitive, has a number of advantages. Firstly, it removes the complexity of specifically modelling every single possible coalition of agents as variable or function nodes, as they are encoded within the utility functions. Secondly, it reduces the possibility of cycles, which simplifies computation by removing redundancy. A formulation similar to this one can be seen in the work of (Waldock et al., 2008); in which a number of stationary sensors are required to coordinate to track moving targets. In our formulation, however, the targets are stationary and the agents move to them.

This formulation is not without issue, however. In this case, a major question is where to situate computation of utility functions. Obviously the tasks themselves cannot compute these values, so the computation will need to be distributed to the agents. This is easy for functions with unary constraints, as they can be assigned to the agent which is the variable. However, this is more difficult with multiple constraints, as the search space, computation, and message passing would need to be coordinated amongst agents. Next, we formulate this problem of *computation distribution*, which is exactly paralleled by the $R||C_{\max}$ problem discussed in Section 2.4, for which no decentralised algorithms exist. Thus, this is an important problem to solve.

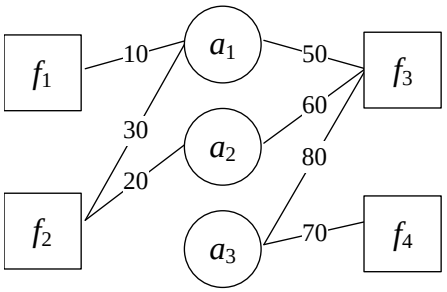


FIGURE 3.2: A graphical representation of the computation distribution problem presented by the example environment in Figure 3.1, in which agents are represented by circles, and functions to be computed by squares. Edges between agents and functions indicate an agent can compute a function, at a cost denoted on the edge.

3.5 Computation Distribution Problem

Given the set of agents $\mathcal{A}$, they must be assigned computation of function nodes, denoted earlier as the set $\mathcal{F}$. An agent a_i is capable of performing the computation of a subset of the function nodes, which we denote as $F_i \subseteq \mathcal{F}$. Similarly, we denote the set of agents that can perform the computation of function f_j as A_j. Each agent a_i may perform computation for any subset of the functions in A_i.

For each agent $a_i \in \mathcal{A}$, we denote a cost function, $\chi_i : F_i \to \mathbb{R}^+$, that returns the total run–time, in seconds, incurred at a_i to perform computation and communication for some function node $f_j \in F_i$. Thus, $\chi_i(f_j)$ returns the application-specific time required for agent a_i to compute and communicate for function f_j. Note that this cost function model is not specific to FMS — in order to create a cost function specific to FMS, message propagation times between function and variable nodes would need to be taken into account, and provision made for cycles in the graph increasing this time. As this is initial work, we have abstracted this cost up to a generalised cost function, and assume that the cost function will be explored in future work. As we will explain later on, this abstraction allows us to provide a solution to both our own computation distribution problem and a well established operations research problem known as scheduling jobs on parallel machines.

A graphical representation of an example computation distribution environment is given in Figure 3.2, in which there are 3 agents (circles) and 4 functions to be computed (squares). Each agent is connected to the functions it can potentially compute for by edges in the graph, and edges are labelled with $\chi_i(t_j)$. Thus, for example, agent a_1 will incur a runtime of 30 to compute function f_2 whereas agent a_2 will only incur a runtime of 20.

Given this, the problem is to schedule the computation of all of the functions in $\mathcal{F}$ across the agents in $\mathcal{A}$ such that all functions are computed and the makespan (i.e., the maximum total computation time at an agent) is minimised. We formally define this

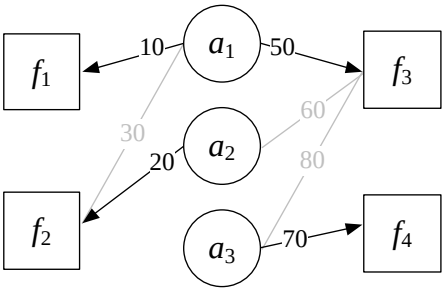


FIGURE 3.3: An optimal mapping from agents to the functions they should compute, for the problem in Figure 3.2. Arrows between agents and functions depict an agent being assigned to compute a function, and greyed out edges indicate other functions an agent could have performed.

objective in the next section.

3.5.1 Objective Function

Specifically, the objective of the computation distribution is to find a mapping $m : \mathcal{A} \rightarrow 2^{\mathcal{F}}$ from agents to functions, such that the makespan is minimised. In particular, we wish to find this mapping subject to a number of constraints. First, each function must only be computed by one agent, since sharing computation of functions would be complicated and introduce additional communication:

$$m(a_i) \cap m(a_j) = \emptyset, \forall a_i, a_j \in A, i \neq j$$

and second, all functions must be computed:

$$\bigcup_{a_i \in A} m(a_i) = \mathcal{F}$$

in which $m(a_i)$ denotes the set of functions assigned to agent a_i, under mapping m. Given this, our objective is to find a mapping m^* as follows:

$$m^* = \arg \min_{m \in M} \max_{a_i \in A} \sum_{f_j \in m(a_i)} \chi_i(f_j) \tag{3.4}$$

where M is the set of all possible mappings.[1] For instance, Figure 3.3 depicts an optimal mapping of the problem in Figure 3.2 where optimal assignments from agents to the functions they should compute are shown with arrows. Thus, the optimal mapping m^* is defined as: $m^*(a_1) = \{f_1, f_3\}$, $m^*(a_2) = \{f_2\}$ and $m^*(a_3) = \{f_4\}$ with a makespan value of $\max(10 + 50, 20, 70) = 70$.

Now, in order to solve the objective function given in Equation 3.4 in a decentralised

[1]Note that we do not model this as a max-product problem where we would be aiming to maximise fairness across agents; rather, we wish to minimise the finish time where all agents compute in parallel.

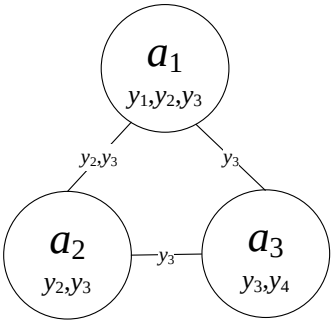


FIGURE 3.4: The junction graph formulation of the scenario given in Figure 3.2. Large circles are cliques, with the elements of the cliques listed. Edges are labelled with common variables between cliques.

way, we propose a solution that relies on decomposing the problem and modelling it as a junction graph, such as the one shown in Figure 3.4, to facilitate the application of a GDL algorithm (since we showed earlier how GDL algorithms are suited to our domain) to solve the problem. Hence, in the next section, we describe how to do so.

3.5.2 Junction Graph Representation

In this section, we show how to represent the computation distribution problem as a junction graph (Jensen, 1996). This representation allows us to decompose our objective function amongst agents so that computation can be distributed throughout the system. In what follows, we give some background on junction graphs. In more detail, a junction graph is an undirected graph with nodes representing *cliques*, which are collections of variables, and edges which represent the overlaps between cliques. Thus, in our junction graph formulation:

- each agent a_i has a node i, known as a *clique*, which represents a collection of variables, $Y_i = \{y_k | f_k \in F_i\}$, from the set of all variables, $\mathcal{Y} = \{y_1, \ldots, y_{|\mathcal{F}|}\}$, which includes one variable y_k for each function $f_k \in \mathcal{F}$.
- each clique node i in the junction graph is associated with a potential function, $\psi_i : Y_i \rightarrow \mathbb{R}^+$, which is a function defined over the set of variables in the clique.
- two clique nodes i and j are joined by one edge that contains the intersection of the variables they represent, i.e. $Y_i \cap Y_j$.

In order to apply min-max, we reformulate the objective function in Equation 3.4 in terms of a junction graph. Figure 3.4 depicts the junction graph representing the problem in Figure 3.2. Thus, the junction graph in Figure 3.4 contains four variables, $\{y_1, y_2, y_3, y_4\}$, which correspond to the four tasks in Figure 3.2. Each variable $y_k \in Y$ takes a value from its domain, Δ_k, which contains all of the indices of agents that can compute function f_k; or, more formally, $\Delta_k = \{i | a_i \in \mathcal{A}, f_k \in F_i\}$. Hence, $y_k = i$ means

that agent a_i is allocated to compute for function node f_k. For instance, in Figure 3.4, the domain of y_2 is composed of two values, 1 and 2, corresponding to the indices of the agents that can compute for function f_2, a_1 and a_2. Notice that, in doing this, we enforce the constraint that exactly one agent must compute for every function.

With slight abuse of notation, we use $\mathbf{Y}_i$ to denote a configuration of the variables in Y_i (i.e. an assignment of each variable in Y_i to a value within its domain). Given this, in our formulation, an agent a_i's clique will contain all variables in Y_i (in Figure 3.4, labels within circles denote agents' cliques). Thus, in Figure 3.4, the set of variables corresponding to agent a_2's clique, Y_2, is composed of y_2 and y_3, which are the two functions that a_2 can compute in Figure 3.2.

Finally, we encode the cost function of agent a_i as a potential function, $\psi_i(\mathbf{Y}_i)$, representing the total time that a_i will take to compute for the functions in the configuration $\mathbf{Y}_i$. Formally:

$$\psi_i(\mathbf{Y}_i) = \sum_{y_k \in \mathbf{Y}_i, y_k = i} \chi_i(f_k) \tag{3.5}$$

Thus, in Figure 3.4 the potential function of agent a_2, ψ_2, which is defined over variables y_2 and y_3, returns a runtime of 60 for the configuration $y_2 = 1, y_3 = 2$, which is the runtime incurred at a_2 to compute function f_3 in Figure 3.2.

By the definition of a junction graph, two agents, a_i and a_j, will be joined by an edge in the junction graph if and only if $Y_i \cap Y_j \neq \emptyset$. In Figure 3.4 edges are labelled with the intersection of two cliques. Thus, agent a_2 is linked to a_3 by an edge that contains the only common variable in their cliques: y_3. Given this, we denote agent a_i's neighbours, $\mathcal{N}(a_i)$, as the set of agents with which a_i shares at least one variable, and therefore, are neighbours in the junction graph. As a result, the junction graph encodes our objective function (Equation 3.4), where m^* and $\mathbf{Y}^*$ are equivalent (so if $f_j \in m^*(a_i)$, then $\{y_j = i\} \in \mathbf{Y}^*$, for all $f_j \in \mathcal{F}$), as follows:

$$\mathbf{Y}^* = \arg\min_{\mathbf{Y}} \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i) \tag{3.6}$$

where $\mathbf{Y}_i$ is the *projection* of $\mathbf{Y}$ over the variables in Y_i. In more detail, given a set of variables $Y_i \subseteq Y$, a projection of $\mathbf{Y}$ over Y_i contains the variable values found in $\mathbf{Y}$ for all $y_k \in Y_i$. For example, if $\mathbf{Y} = \{y_1 = 1, y_2 = 4\}$ and $Y_1 = \{y_2\}$, then $\mathbf{Y}_i = \{y_2 = 4\}$.

3.5.3 Decomposing the Objective Function

Now that we have a junction graph formulation of the problem, which has allowed us to decompose our objective function into $|A|$ local functions, we can distribute the computation of our objective function amongst the agents. In order to do this, we compute the marginal function $\zeta_i(Y_i)$ at each agent, which describes the dependency of

the global objective function (given in Equation 3.4) on agent a_i's clique variables. This is computed by each agent a_i as follows:

$$\zeta_i(\mathbf{Y}_i) = \min_{\mathbf{Y}_{-i}} \max_{a_j \in \mathcal{A}} \psi_j(\mathbf{Y}_j) \tag{3.7}$$

where $Y_{-i} = Y \setminus Y_i$ and $\mathbf{Y}_j$ is the projection of $\mathbf{Y}_{-i}$ over the variables in Y_j. Thus, the marginal function for agent a_3 in Figure 3.4 with $Y_3 = \{y_3, y_4\}$ is computed as $\zeta_3(y_3 = 1, y_4 = 3) = min(70, 90)$, $\zeta_3(y_3 = 2, y_4 = 3) = min(70, 100)$,$\zeta_3(y_3 = 3, y_4 = 3) = min(130, 150)$.

Finally, in the presence of a unique solution, the optimal state of a_i's clique variables is:

$$\mathbf{Y}_i^* = \arg\min_{\mathbf{Y}_i} \zeta_i(\mathbf{Y}_i) \tag{3.8}$$

Thus, in Figure 3.4, a_3's clique variables have two optimal states $\mathbf{Y}_3^* = \{(y_3 = 1, y_4 = 3), (y_3 = 2, y_4 = 3)\}$. This decomposition facilitates the application of the min-max GDL algorithm to find a solution to $R||C_{\max}$ in a decentralised fashion. Thus, later on, in Chapter 5, we use min-max as part of our solution to the computation distribution problem, and prove its most important property: that with min-max operators, GDL algorithms are guaranteed to converge in any junction graph within a finite number of iterations.

3.6 Summary

In this chapter, we have shown how we formalise a generic dynamic task allocation problem as a DCOP. In particular, we have chosen to formalise the problem with agents as variable nodes, and tasks as factor nodes. This formalism presents a significant question, with regard to where to situate computation of task utility functions. Thus, we also provided a formulation of this problem of computation distribution, and showed how to formalise the computation distribution problem as a junction graph, with functions to be computed as variables, and agents to perform the computation as owning clique nodes containing those variables.

In the next chapter, we present our FMS, BnB FMS, and BFMS algorithms running over the formulation detailed in this chapter. These algorithms reduce the communication and computation performed by the max-sum algorithm (see Section 2.2) in environments which change over time. Later, in Chapter 5, we present ST-DTDA, which is our algorithm for solving the computation distribution problem which we defined in this chapter.

Chapter 4

Decentralised Algorithms for Dynamic Task Allocation

In this chapter, we describe three decentralised algorithms for dynamic task allocation: FMS, which makes savings in communication and computation after a change in the graph, BnB FMS, which reduces the computational complexity of FMS, and BFMS, which provides quality bounds on the solutions it provides.

Thus, we first describe FMS, which reduces the computation and communication performed by the max-sum algorithm when changes are made to the environment it runs in (described in Section 2.3.2) through the use of local storage and decision rules. In so doing, it can address coordination problems in rapidly changing environments, such as those presented by disaster management scenarios.

Second, we present BnB FMS, which specifically reduces the computational complexity of FMS through the use of a novel online domain pruning algorithm, and branch and bound search trees within function nodes. In so doing, we show that, in practice, the computation and communication used by FMS is significantly reduced.

Third, we present BFMS, which uses principles from BMS (see Section 2.3.5) in order to provide bounded approximate solutions on graphs containing cycles. To do this, we have modified the GHS algorithm (used by BMS to find the MST, see Section 2.3.4) to recompute smaller sections of the graph in order to reduce overheads, while still improving the graph. We call this modified algorithm iGHS.

Now, the application of all three of our algorithms comes at the expense of linear storage at each agent — however, this is minimal. Nevertheless, when deciding between the algorithms, BFMS should be used where bounded solutions are absolutely necessary, at the expense of some additional preprocessing and communication, BnB FMS should be used for unbounded solutions in large-scale environments, again at the expense of some

additional preprocessing and communication, and FMS should be used for unbounded solutions in smaller scale environments.

We begin the chapter by reiterating some basic definitions from Chapter 3.2, expanding on some of them to facilitate description of our algorithm. We then provide a description of FMS, BnB FMS, iGHS and BFMS and prove their properties in turn. Finally, we provide an empirical evaluation of FMS, BnB FMS and BFMS.

4.1 Basic Definitions

Here we recall some basic definitions from Section 3.2. Our domain consists of a set of agents $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$, and a set of tasks $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$. Utility is gained by an agent when a task is performed — we model this as a task generating some utility based on the agents assigned to it. More formally, we denote the utility generated by task t_j when the group of agents (coalition) C_j are assigned to it as $V(C_j, t_j)$.[1]

This domain is then formulated as a factor graph as described in Section 3.4, where agents are the variable nodes and the task utility functions are the function nodes. We define the task utility function of task t_j as f_j, and the variables to that function as $\mathbf{x}_j$.

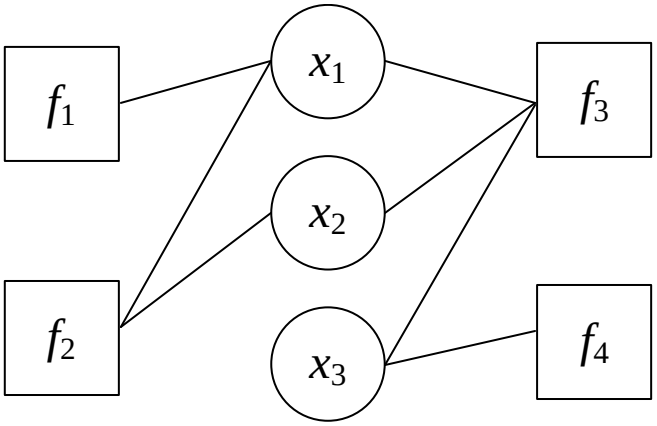


FIGURE 4.1: An example factor graph, with agents as variables (circles), tasks as functions (squares), and edges denoting where variables are parameters to functions.

We define a *disruption* in the graph as the addition or removal of a task from $\mathcal{T}$ which results in the addition or removal of a factor from the factor graph. This may happen for a number of reasons. First, a task t_i may be removed from $\mathcal{T}$, for example when it has been completed or becomes impossible to complete. Second, a task t_i may be added to $\mathcal{T}$ when new information is received (e.g., when a new task is discovered). Third, if the wrong information is received about a task t_i (e.g., false information has been

[1] We assume that the computation of this value is known at all agents — a simple example of this function would be for a coalition C_j to generate utility $|C_j|$, a fact which would be known at all agents.

obtained), then it will be removed from $\mathcal{T}$. In the max-sum algorithm, when a factor is added to or removed from the underlying factor graph, there is no specific protocol to follow — thus, the assignments in the affected part of the graph must be recalculated. This means that the variables and factors must begin propagating messages (found in Section 2.3.2, Equations 2.2 and 2.3, respectively) in order to find the best assignment of variables to values. These messages will be sent from and to all affected variable nodes, including variable nodes which are not able to find a better assignment than the one they had before the disruption. Nodes such as these will, therefore, get assigned back to the same value. In terms of Figure 4.1, if x_1 was assigned to f_2, and f_1 was removed from the graph, under the max-sum algorithm, x_1 would need to propagate messages again in order to compute that it should be assigned to f_2, as it was before. However, this saving is difficult to characterise in formulae because the amount of messages saved depends on the structure of the factor graph, the initial variable assignments, and the particular change being made to the graph — thus, this is why we perform an empirical evaluation of our algorithms later on in Section 4.5. Against this background, in the next section we introduce FMS, an algorithm that builds upon max-sum to better cope with disruptions in the factor graph.

4.2 Fast-max-sum

The FMS algorithm extends the standard max-sum (see Section 2.3.2) in two main ways. First, in order to reduce the number of states over which each factor has to compute its solution, we introduce new functions on variable and factor nodes that single out the states that matter to them, thus significantly reducing communication and computation. Second, we introduce new functionality that allows each agent to decide when to send messages to its other connected factors when changes happen in the factor graph (i.e., a factor is removed or added), at the expense of a linear amount of storage at each agent. We detail our algorithm and these two extensions in the following subsections and show how they allow us to make significant computational and communication savings in Section 4.5.

4.2.1 The Algorithm

In Algorithm 4.1, we give the procedure run at a function node, f_j, when a message is received from a variable node. In more detail, when the function receives a message, it first checks if the message is different from the last message it received from that variable (line 2), and, if it is, then the function recomputes and sends messages to all of its neighbours (line 4). Similarly, when a variable receives a message from a function node (as shown in Algorithm 4.2) it first checks if its z^τ values have changed (line 2) since the previous timestep, and, if they have not, then the variable knows it does not

Algorithm 4.1 When a message is received at factor f_j

Received $q_{i\to j}$:
if $q_{i\to j}$ has changed **then**
  `// Send message to all neighbours`
  Send $r_{j\to k}$ to all $k \in \mathcal{N}_j$
end if

Algorithm 4.2 When $r_{j\to i}$ is received at a variable.

Require: $z_i^\tau(\cdot)$
Received $r_{j\to i}$ at time τ':
if $z_i^\tau(x_i) = z_i^{\tau'}(x_i)$ where $x_i = t_j$ and where $x_i \neq t_j$ **then**
  Send no further messages.
else
  if $r_{k\to i}$ received from all $k \in \mathcal{M}_i$ **then**
    $x_i \leftarrow \arg\max_{x_i} z_i^\tau(x_i)$ `// Once received messages from all neighbours, assign variable to value with highest utility`
  else
    Send $q_{i\to k}$ to all $k \in \mathcal{M}_i$ `// Continue message propagation`
  end if
end if

need to send any further messages (line 3). However, if either of the values have changed, and the variable has received messages from all of the functions incident on it in the factor graph (line 5), then the variable chooses a new state (line 6). If, however, the variable has not yet received messages from all functions incident on it, then it continues message propagation (line 8).

4.2.2 Reducing Communication and Computation

In order to reduce the number of states each factor needs to compute its solution over, we simplify the domain of each variable communicated to each connected factor to only two states. These states represent the fact that an agent is assigned to a specific task (the factor) or not, as opposed to every possible state of the variable. With this change, we can now specialise the message computation performed by the original max-sum which applies over all states of the variables involved (see Equations (2.2) and (2.3) in Section 2.3.2). Hence, in what follows, we introduce new functions to manage these new messages that inform the factors of these states and show why FMS retains the same properties as max-sum with this reduction in state space.

In order to do this, we use an indicator function $I_{ij}(x_i) \in \{0, 1\}$ which returns 1 if $x_i = t_j$, and 0 otherwise, or, more formally:

$$I_{ij}(x_i) = \begin{cases} 1 & \text{if } x_i = t_j, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

Thus, using this new function we reduce the messages between variables and functions, which are sent as described in Algorithms 4.1 and 4.2, to contain two values, as follows:

From variable to function:

$$q_{i\to j}(I_{ij}(x_i)) = \begin{cases} \displaystyle\sum_{k\in\mathcal{M}_i\setminus j} r_{k\to i}(0) & \text{if } I_{ij}(x_i) = 1, \\ \displaystyle\max_{b\in\mathcal{N}(i), b\neq j} \left[r_{b\to i}(1) + \sum_{k\in\mathcal{M}_i\setminus b,j} r_{k\to i}(0) \right] & \text{otherwise.} \end{cases} \tag{4.2}$$

where $\mathcal{M}_i$ denotes the set of indices of functions connected to x_i in the factor graph. The above message, sent in line 8 of Algorithm 4.2, differs from the usual max-sum message (see Equation (2.2) in Section 2.3.2) in that the variable only sends the utility values for two states; where the agent a_i assigns itself to t_j (i.e., where $I_{ij}(x_i) = 1$) and where it does not (i.e., where $I_{ij}(x_i) = 0$). This is possible here because the utility gained by function f_j by assigning a_i to t_k for $k \neq j$ is the same for all k since the agent does not help in completing t_j in this case. Hence, what matters to f_j is only the utility that the rest of the system gets for *not* assigning the agent to t_j.

From function to variable:

$$r_{j\to i}(I_{ij}(x_i)) = \max_{\mathbf{x}_j \setminus i} \left[f_j(\mathbf{x}_j) + \sum_{k \in \mathcal{N}_\mathbf{j} \setminus i} q_{k\to j}(I_{kj}(x_k)) \right] \tag{4.3}$$

where $\mathcal{N}_j$ denotes the set of indices of variables connected to f_j in the factor graph. The above, sent in line 4 of Algorithm 4.1, differs from the usual max-sum message (see Equation (2.3) in Section 2.3.2) in that the factor does not need to compute the utility the system gets for all values of x_i. Instead, it only computes for a_i being assigned to t_j or not (i.e., for the two values of $I_{ij}(x_i)$). When a_i is not assigned to t_j, $f_j(\mathbf{x}_j)$ is independent of the specific task that the agent is assigned to and hence we can generalise this allocation to be any task other than t_j. Note that this operation is different from the one in max-sum which would have searched assignments of the variable which do not improve the utility of the factor in any way. Thus, we effectively prune the space that would have originally been searched by max-sum without losing any information.

Since now a variable node receives only two values per factor it is connected to,[2] the variable node needs to sum these messages in a different way from max-sum. Basically, when a variable node x_i has received $r_{j\to i}(I_{ij}(x_i))$ for $I_{ij}(x_i) = 0$ and $I_{ij}(x_i) = 1$, the variable simply adds $r_{j\to i}(1)$ to all other messages $r_{k\to i}(0)$. Hence the computation of the function $z(\cdot)$ is now (compared to Equation (2.4) in Section 2.3.2):

$$z_i(x_i) = \left(r_{j\to i}(1) + \sum_{k \in \mathcal{N}_i \setminus j} r_{k\to i}(0) \right) \tag{4.4}$$

given $x_i = t_j$. In so doing, we get the total utility for all states of the variable. Then, the variable can choose which value it takes as $\arg\max_{x_i} z_i(x_i)$ as before. Here, we are not only making a communicational saving but also using a more natural problem representation by only sending clearly defined binary values — if we were to send a matrix of values for all states, as in max-sum, in practice we would have to concern ourselves with how best to index the matrices to obtain values for each state.

Note that, since FMS does not specifically try to reason about cycles in the factor graph, FMS cannot be guaranteed to converge on graphs with cycles similar to max-sum.[3] However, both FMS and max-sum are proven to converge on acyclic graphs (Aji and McEliece, 2000). We next detail how FMS adapts to disruptions.

[2] This could actually be reduced to only one value if the nodes were only to send the difference between their values for the two states.

[3] We assume that the algorithm has converged after no messages have been received at an agent for a pre-determined length of time. This time will differ depending on the size of the environment and the latency of the communication links amongst the agents, to ensure that the agents do not falsely assume the algorithm has converged when it has not.

4.2.3 Managing Disruptions

As we mentioned in Section 4.1, when we refer to a *disruption*, we mean a change in the domain of one or more variables in the underlying factor graph. Now, in order to define how the algorithm chooses to send a message, we first assume that at time τ, the allocation computed by each variable is $\arg\max_{x_i} z^{\tau}(x_i)$. Then, at time τ' two types of disruption may happen:

1. A new task t_k appears at $\tau' > \tau$ — the set of tasks that needs to be assigned is augmented to $\mathcal{T}^{\tau'} = \mathcal{T}^{\tau} \cup \{t_k\}$. This also means that the set of indices of factor nodes is increased to $\mathcal{M}^{\tau'} = \mathcal{M}^{\tau} \cup \{k\}$.
2. An existing task t_k disappears at time $\tau' > \tau$ — the set of tasks that needs to be assigned is reduced to $\mathcal{T}^{\tau'} = \mathcal{T}^{\tau} \setminus t_k$. This also means that the set of factor nodes is decreased to $\mathcal{M}^{\tau'} = \mathcal{M}^{\tau} \setminus \{k\}$.

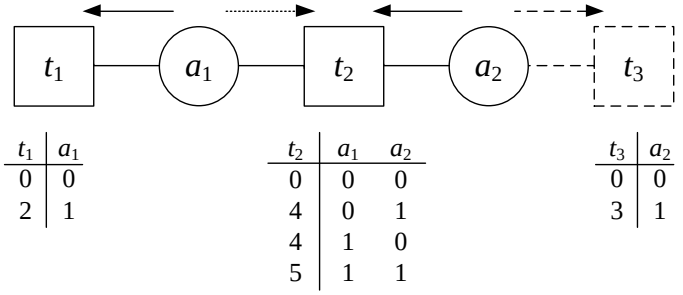
Given the above, the domains of the variables will also change accordingly. The result is that the variables x_i with $t_k \in D_i$ will need to make different decisions based on whether the factor is removed or has just been added. Essentially this means that:

1. if f_k is added, then $z^{\tau'}(x_i) = z^{\tau}(x_i) + r_{k \to i}(I_{ik}(x_i))$.
2. if f_k is removed, then $z^{\tau'}(x_i) = z^{\tau}(x_i) - r_{k \to i}(I_{ik}(x_i))$.

Then, the decision that the agents make is as follows. Assuming for each f_k for $t_k \in D_i$ we define the utility for $I_{ik}(x_i) = 1$ and $I_{ik}(x_i) = 0$ as $\{z^{\tau}(x_i = t_k), z^{\tau}(x_i \neq t_k)\}$, then:

- If both $z^{\tau}(x_i = t_k) = z^{\tau'}(x_i = t_k)$ and $z^{\tau}(x_i \neq t_k) = z^{\tau'}(x_i \neq t_k)$, then x_i does not need to transmit a newly computed $q_{i \to k}(I_{ik}(x_i))$ based on the new domain of x_i. This is shown in lines 2 and 3 of Algorithm 4.2.
- Otherwise, x_i has to send the message (line 4 onwards, Algorithm 4.2). This is because, if the utility that a_i achieves by being allocated (or not) to t_k changes, then the system's utility might change as well and hence messages need to be sent around in any case.

Note that here we use relatively simple filtering to reduce the *number* of messages sent — there is space for some further improvements to be done here by using thresholds (i.e., if a value changes only a small amount then do not resend the message). However, these thresholds would be difficult to compute in practise since their values would depend on other variables and functions either directly or indirectly related to each variable, so there would need to be some trading off between threshold accuracy and the complexity of the threshold computation.



t_1	a_1
0	0
2	1

t_2	a_1	a_2
0	0	0
4	0	1
4	1	0
5	1	1

t_3	a_2
0	0
3	1

FIGURE 4.2: Example of a new task t_3 being discovered in the system. The task is added (shown by the dotted line) to an existing factor graph. Then a_2 needs to decide whether it sticks to its current assignment, t_2 (denoted by the solid arrow), or changes to t_3 (denoted by the dotted arrow). The tables show the utility (in the left-most column) for the assignment of each agent to each task. At time τ only the tables for t_1 and t_2 exist and at time τ' the table for t_3 is introduced.

For example, consider Figure 4.2 that shows a situation with two agents a_1, a_2 and two tasks t_1, t_2 at time τ. Suppose at time $\tau' > \tau$ a third task t_3 is discovered and so the factor graph changes as shown in the figure. The tables represent the utility functions for each task. The assignment of an agent to the task is noted in binary format to represent whether it is allocated (1) or not (0). The value in the left-most column represents the utility obtained for the assignment selected (e.g., $\{a_1 = 0, a_2 = 1\}$ results in a utility of 4 at f_2).

The best allocation at time τ is computed as $\{a_1 = t_1, a_2 = t_2\}$ which gives a value of $4+2 = 6$, while when the new task is discovered then the best allocation is $\{a_1 = t_2, a_2 = t_3\}$ which gives a value of $4 + 3 = 7$. The change in the factor graph will result in a different $z_2^{\tau'}(a_2)$, which will trigger information propagation leading FMS to change the allocation and obtain the best value. In particular, notice that when t_3 is discovered we have $z_2^{\tau}(a_2 = t_2) = 4$ and $z_2^{\tau}(a_2 \neq t_2) = 0$ while $z_2^{\tau'}(a_2 = t_2) = 4$ and $z_2^{\tau'}(a_2 \neq t_2) = 3$, which results in $z_2^{\tau}(a_2 = t_2) = z_2^{\tau'}(a_2 = t_2)$ but $z_2^{\tau}(a_2 \neq t_2) \neq z_2^{\tau'}(a_2 \neq t_2)$ and message propagation is necessary to deal with the change in the factor graph and reach a better allocation.

In the case where the utility does not change (i.e., $z_2^{\tau}(a_2 \neq t_2) = z_2^{\tau'}(a_2 \neq t_2)$) with respect to t_k, then FMS prevents any message from being sent (see the rules above) as the rest of the system (starting from f_k) will not be affected (since a_2 cannot do any better by changing its assignment). Instead, in max-sum, any change in utility in any state of the variable initiates a new message (see Equation (2.2) in Section 2.3.2) and hence results in it recomputing Equation (2.3) in Section 2.3.2 (which maximises over all states of all its connected variables) only to find out that the allocation does not change. Comparatively FMS only needs to check over t_k's states to decide whether to send a message or not, instead of checking over all states of its connected variables, thus minimising the computation needed to implement any changes in utility. Thus, by distributing some of the computation on the variable nodes (in addition to factor

nodes) and filtering out messages, FMS can avoid both the redundant computation and messages of max-sum. However, later on in our evaluation we show that the biggest computational savings actually come from the reduction of message size and computational complexity, as opposed to from the message filtering.

Similar to variable nodes, we also introduce storage at function nodes. In more detail, we store the last received message from each variable connected to each function in the factor graph, so that a function node need only send a new set of messages if it has received new information (line 2, Algorithm 4.1). Whilst this comes with the downside of needing additional storage for each function node, the storage needed is linear. Another type of disruption that our algorithm could face is a disruption in communication — for example, if a lossy communication line were used, then some messages could be lost before they reached their destination. In such cases, variable and function nodes will keep re-sending their messages at regular intervals, to ensure that all receiving nodes have the most up-to-date information. Evidently, this comes at the cost of extraneous communication being used, and the trade-off between additional bandwidth and solution accuracy should be considered in scenarios with unreliable communication.

Also note that FMS and max-sum have essentially the same behaviour when it comes to computing the solution since they both take into account the same information (i.e., changes in utility that will affect the assignment of agents to tasks). FMS is simply more efficient at doing so. Thus, if efficiency in responding to changes in the environment is required, then FMS should be used. However, if storage is unavailable and efficiency is less important, then max-sum should be used.

Now that we have explained our algorithm, we provide a worked example of its execution in the next section.

4.2.4 Worked Example

This section contains a worked example of the FMS algorithm. We base this on the example scenario shown in the factor graph in Figure 4.1. We repeat the factor graph diagram below, in Figure 4.3 for ease of reading; we have added contributions to the graph so an additive utility function can be demonstrated.

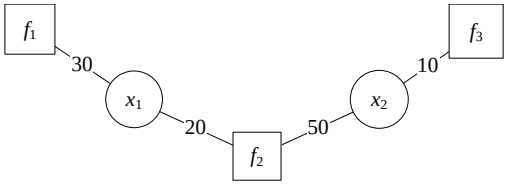


FIGURE 4.3: The example from Figure 3.1 formulated as a factor graph, with agents as variables (circles), and tasks as factors (squares). Numbers on lines joining agents to tasks represent the contribution of each agent to each task.

In our scenario, we use the utility function given in Section 4.1.

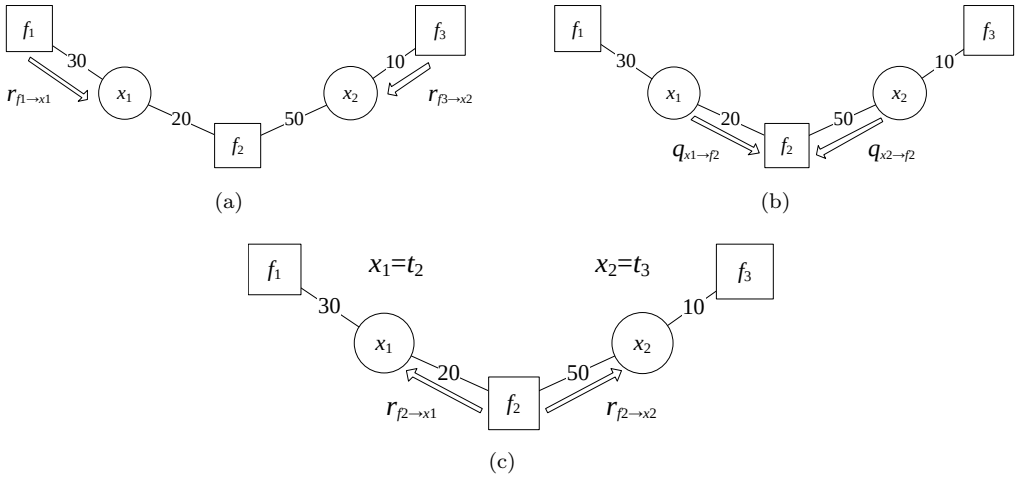


FIGURE 4.4: Example execution: stage 1.

Figure 4.4 shows the initial execution of the algorithm. In Figure 4.4(a), we see f_1 and f_3 sending their $r_{j\to i}$ messages (as defined in Equation 4.3) to x_1 and x_2, respectively. On receipt of these messages, x_1 and x_2 store their current value of $z_a(\cdot)$ (see Equation 4.4), and the next step of the execution begins. As shown in Figure 4.4(b), x_1 and x_2 send their $q_{i\to j}$ messages to f_2 (as defined in Equation 4.2). f_2 uses these messages to calculate and send its $r_{j\to i}$ messages, as shown in Figure 4.4(c). On receipt of these messages, x_1 and x_2 store their values of $z_a(\cdot)$. As x_1 and x_2 have now received messages along all of their edges, a decision can be made as to their assignment. x_1 receives the most utility being assigned to f_2, and x_2 receives the most utility being assigned to f_3, and so these assignments are stored in the agents' x_i variables.

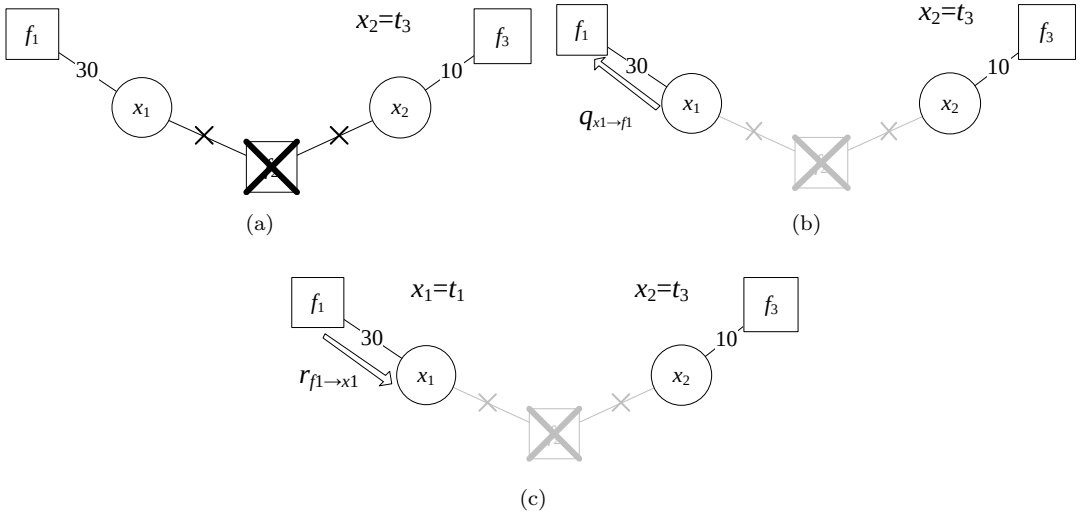


FIGURE 4.5: Example execution: stage 2. Factor f_2 has been removed.

Now, we can remove f_2 from the factor graph, for example after x_1 arrives there and finds it empty, which removes it from the list of tasks x_1 must complete. The execution

of the algorithm then proceeds as in Figure 4.5. In Figure 4.5(a), we see that x_1 has removed its target (which was t_2), as it no longer exists in the factor graph. As x_1 now has no target, it must find a new assignment, and therefore sends $q_{i\to j}$ in Figure 4.5(b). Once x_1 receives an $r_{j\to i}$ message from f_1, it has received a message on every edge, and therefore can set $x_1 = t_1$, as shown in Figure 4.5(c). Meanwhile, the value of x_2, t_3, still exists in the factor graph. x_2 compares f_3's utility with the previous utility it held for f_3, which has not changed. Thus, x_2 does not need to find another assignment, and f_3 does not need to send any messages in 4.5(c).

It can be seen from the example that FMS removes the unneeded communication and computation that would be incurred if the max-sum algorithm were to be used in the same situation (as discussed in Section 4.2.3). By introducing storage at each variable, we allow variables to maintain their previous assignment if they will not benefit from changing assignment after a disruption in the graph.

Given all this, in the next section we discuss how FMS meets the key properties of dynamic algorithms, as highlighted in Section 2.2.5.

4.2.5 Properties of FMS

As discussed earlier (Section 2.3.2), max-sum does not have any defined behaviour after a change in the environment. Thus, it is not possible to prove that max-sum is or isn't superstabilizing — in fact, to make max-sum superstabilizing it would be necessary to store previous states and react to changes in the environment in the way that FMS does. Given this, in this section we show that FMS is superstabilizing (see Section 2.2.5.1), subject to the following predicates: *legitimacy*: $U(\mathbf{x}) = \max_{\mathbf{x}} \sum_{t_j \in \mathcal{T}} f_j(\mathbf{x}_j)$, on acyclic graphs, where $U(\mathbf{x})$ is the total utility of assignment $\mathbf{x}$, and *passage*: the previous assignment of variables is maintained until a new solution has been computed.

Proposition 4.1. *FMS is superstabilizing on tree structured graphs, with respect to the legitimacy and passage predicates given above.*

Proof. First, FMS is an extension to max-sum which is proven to converge to an optimal solution on tree structured graphs in a finite number of messages (Mackay, 2003). Second, when a change occurs in the graph, FMS is run again, and therefore, provided the change did not introduce a cycle into the graph, FMS is guaranteed to converge to the optimal again, satisfying the legitimacy predicate, thus reaching a legitimate state within a finite amount of time. Now, FMS is superstabilizing because it maintains each variable's previous state at each variable node at all times, only updating it after recalculation, and so, the passage predicate always holds. □

In addition to this, FMS is anytime (see Section 2.2.5.2), because the algorithm can be stopped at any time during its execution and a solution will exist (and can be computed

	$\lvert\mathcal{T}\rvert$				
δ_v	10	50	100	500	1000
2	40	200	400	2000	4000
3	80	400	800	4000	8000
4	160	800	1600	8000	16000
5	320	1600	3200	16000	32000

TABLE 4.1: Average total state space explored with increasing graph density and number of tasks.

using the marginal functions at each agent). We cannot give any theoretical results as to the competitiveness of FMS, since a change in the environment could, in the worst case, cause all messages to be resent, and all variables to be assigned different values. In this work, we consider competitiveness in terms of variable reassignments, since changing the value of a variable (and so, changing an agent's assignment) is likely to carry some penalty. Specifically, we assume that over some finite time horizon, the optimal solution in terms of competitiveness, given full knowledge of the changes that are to happen, would assign only the variables present at the final timestep to their optimal values at the final timestep, thus incurring no variable reassignment penalties.

4.2.6 Summary

To summarise, by only considering $x_i = t_k$ and $x_i \neq t_k$, FMS reduces the communicated domain size of each variable from d to 2, compared to a standard implementation of max-sum. Hence, the computational complexity of a factor with n variables of domain size d in FMS is $O(2^n)$, in contrast to $O(d^n)$ required by naïvely applying max-sum to the same environment. In order to do this, FMS requires storage of a number of values in order to operate, which can be represented as follows:

$$\mathcal{S} = 2|\mathcal{A}||\mathcal{E}| \tag{4.5}$$

where $\mathcal{S}$ is the total storage needed, and $\mathcal{E}$ is the set of edges per variable. This demonstrates that the storage that FMS requires is linear in the number of edges and agents. We can translate this value into bytes by multiplying it by 8 (there are 8 bytes in a double precision value) in order to demonstrate how much storage our algorithm will use — for example, an environment with 100 agents, each with 20 tasks to choose between will require $2 \times 100 \times 20 \times 8 = 31.25$ kilobytes of storage in total (i.e., across all agents). This value is feasible for reasonably large environments, especially if we assume that each agent is equipped with a smartphone, which generally have at least a megabyte of storage space, if not more.

Nevertheless, even with the reduction in the computation needed at each factor, the size of the state space explored is still exponential in the number of agents that can be assigned to each task. This means that, even though the total state space explored by all

factors scales linearly in the number of tasks, as the density of tasks and agents (i.e., the number of agents connected to each task) increases, the total state space will increase very quickly. The overall worst case computational complexity is $\mathcal{O}(|\mathcal{F}| \times 2^{\delta_{\max}})$, where $\delta_{\max}$ is the maximum arity of a function — i.e., the maximum number of agents that could complete a task in the environment. We illustrate this in Table 4.1, which contains the computed average state space for $|\mathcal{T}| = \{10, 50, 100, 500, 1000\}$, and demonstrates the growth of the overall state space as the density and number of tasks increase. This table shows there is a need to attempt to reduce the total state space explored to ensure that the algorithm runs as quickly as possible in large-scale environments. Hence, in the next section, we present our branch and bound approach to reducing the state space.

4.3 Scaling up FMS

In order to combat this key bottleneck of FMS and enable it to scale effectively in the number of agents and tasks, it is crucial to use techniques that reduce the size of the state space explored: by using a preprocessing step to reduce the number of tasks and coalitions considered, whilst ensuring that such techniques remain both effective and efficient as the environment changes. In doing this, we create an efficient, decentralised task allocation algorithm for use in dynamic environments, the use of which is preferable to FMS in large-scale environments.

4.3.1 BnB FMS

Our approach is similar to that of BnB MS, which we described in Section 2.3.3. However, the key difference is that we explicitly cater for environments that change over time, whereas BnB MS doesn't specifically cater for such environments, and as such would need to be completely re-run after every change in the environment, as we show later. Moreover, BnB MS is based on max-sum and thus incurs additional communication and computation needed to search through full variable domains, as opposed to exploiting the ability to use binary values in task allocation environments, as we showed earlier. Thus, in this section, we present BnB FMS, which addresses the shortcomings of FMS and BnB MS to produce a novel, scalable, dynamic task allocation algorithm. To do this, we remove tasks that an agent should never perform with our online domain pruning (ODP) algorithm, and interleave this with optimising the objective function given in Section 3.2 using branch-and-bound and FMS.

4.3.1.1 Online Domain Pruning (ODP)

As we mentioned earlier, the functions we consider represent the utility gained from using coalitions of agents to perform tasks. This allows us to compute a variable's contribution

to a function for a given task, as part of a particular variable assignment. Now, if we sum the contributions of all $(x_i = t_j)$ in a particular $\mathbf{x}_j$, we obtain $f_j(\mathbf{x}_j)$. Hence, these contribution values can be used to locally compute which variable states (if any) will never provide a significant enough gain in utility. This is achieved by computing exact upper and lower bounds on each variable's contributions.

In more detail, we compute the upper bound on the contribution of x_i to f_j as $\delta_j(x_i)^{ub} = \max_{\mathbf{x}_j} \delta_j(x_i, \mathbf{x}_j)$. Similarly, we compute the lower bound on x_i's contribution to f_j by $\delta_j(x_i)^{lb} = \min_{\mathbf{x}_j} \delta_j(x_i, \mathbf{x}_j)$. Hence, $\delta_j(x_i)^{ub} \geq \delta_j(x_i)^{lb}, \forall f_j \in \mathcal{F}_i, x_i \in \mathcal{X}$. We can compute these bounds for any arbitrary function simply by using brute force, or by analytically deriving specific bounds for the specific value function used. The computational impact of computing these bounds by brute force can be mitigated by caching the bounds for a function so they need only be computed once. This allows bound computation to be less computationally expensive than re-computing messages such as those in Equations 4.3 and 4.2, as the values of messages such as these rely on messages received from others. Thus, whilst there does exist a tradeoff between computing these bounds in order to reduce the complexity of FMS, and simply running FMS, we show in our empirical evaluation that, in practice, running ODP and FMS is much faster than just using FMS.

As mentioned above, contribution bounds can be used to find any variable states which will never give a significant enough gain in utility. We call such states *dominated* states. Thus, in this phase, we use localised message-passing to find and remove dominated states from variable domains. In addition, we recognise that changing variable domains (and addition and removal of variables themselves) may change which states are dominated, and we specifically cater for this by maintaining and updating state information for each variable, only passing new messages when they are absolutely necessary. Moreover, we show that this phase prunes all dominated states and never compromises solution quality.

In more detail, we define dominated variable states as those that can never maximise Equation 3.4, regardless of what others do. Formally, at variable x_i, task t_j is dominated if there exists a task t^* such that:

$$\delta_j(x_i, \mathbf{x}_{-i} \cup (x_i = t_j)) \leq \delta^*(x_i, \mathbf{x}_{-i} \cup (x_i = t^*)), \forall \mathbf{x}_{-i} \in \mathbf{X} \tag{4.6}$$

where $\mathbf{x}_{-i}$ represents the state of all variables excluding x_i, and $\mathbf{X}$ is the set of all possible joint variable states for $\mathbf{x}_{-i}$.

We can decentralise the computation of these dominated states through local message-passing, as described in Algorithms 4.3 and 4.4, where, at a variable, $\mathbf{\Delta}$ represents the stored contribution bounds received from functions, and $\tilde{T}_i$ represents a variable's *modified* domain — i.e., with values removed when pruned. Storing the received bounds and modified domains allows ODP to run online, as a variable can update this domain

Algorithm 4.3 ODP at function f_j

Procedure `sendPruningMessageTo(`x_i`)`
Compute $\delta_j(x_i)^{ub}$ and $\delta_j(x_i)^{lb}$
Send $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$ to x_i.
Procedure: On receipt of `PRUNE` from x_i
$\mathcal{X}_j \leftarrow \mathcal{X}_j \setminus x_i$ `// Remove from domain`
Procedure: On receipt of `ADD` from x_i
$\mathcal{X}_j \leftarrow \mathcal{X}_j \cup x_i$ `// Add to domain`
Run procedure `sendPruningMessageTo(`x_k`)` for all $x_k \in \mathcal{X}_j$

Algorithm 4.4 ODP at variable x_i.

Procedure `startup()`
$\tilde{T}_i = T_i$ `// Initialise modified domain`
$\mathbf{\Delta} = \emptyset$ `// Initialise stored bounds`
Send `ADD` to all f_j where $t_j \in \tilde{T}_i$.
Procedure: On receipt of $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$ message from f_j
if $t_j \notin T_i$ **then**
  $T_i \leftarrow T_i \cup t_j$ `// Add to domain`
  $\tilde{T}_i \leftarrow \tilde{T}_i \cup t_j$ `// Add to modified domain`
end if
if $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle \notin \mathbf{\Delta}$ **then**
  $\mathbf{\Delta}_j = \langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$. `// Update stored message`
  if $\exists \mathbf{\Delta}_k, \forall f_k$ where $t_k \in \tilde{T}_i$. **then**
    while $\exists t_j \in \tilde{T}_i : \delta_j(x_i)^{ub} < \max_{t_k \in \tilde{T}_i} \delta_k(x_i)^{lb}$ **do**
      $\tilde{T}_i \leftarrow \tilde{T}_i \setminus t_j$ `// Remove from domain`
      $\mathbf{\Delta} \leftarrow \mathbf{\Delta} \setminus \mathbf{\Delta}_j$ `// Remove from stored bounds`
      Send `PRUNE` to t_j
    end while
  end if
end if
Procedure: a_i can no longer perform t_j
$T_i \leftarrow T_i \setminus t_j$
if $t_j \in \tilde{T}_i$ **then** `// Removed task has not been pruned`
  Send `ADD` to all f_k where $t_k \in T_i \cap \tilde{T}_i$
  $\tilde{T}_i \leftarrow T_i$ `// Start pruning again`
end if

in response to changes in the environment using both newly- and previously-received bounds (as we will discuss later). This is different to BnB MS, where variable domains would have to be completely restored and the algorithm re-run after every change in the environment, incurring potentially unnecessary overheads.

In more detail, ODP begins at the variables (lines 1–4, Algorithm 4.4), which make a copy of their domains so that they can keep track of their pruned domain as well as their full domain (so that it can be restored after a change, if needed), and announce which tasks they can perform. When a function receives one of these announcements, it will send upper and lower contribution bounds (Algorithm 4.3, lines 6–8), which the variable uses to update its domain (if needed) and stored bounds (lines 5–10, Algorithm 4.4). Next, if the variable is aware of the bounds on its contribution to every task it can perform (i.e., has stored bounds for every task in its domain) then it can begin searching for dominated states. If any dominated states are found, then they are removed from the modified domain, and functions representing those states are informed using `PRUNE` messages (lines 11–16, Algorithm 4.4). In response, the functions will update their own domains. We prove that ODP will never prune an optimal state in Section 4.3.3.

Once the algorithm has completed, BnB FMS begins execution (as described in the next section). However, if x_i detects that it can no longer perform task t_j, then t_j is removed from x_i's domain and updated domain (lines 16–20). If x_i had previously pruned t_j from $\tilde{T}_i$ during ODP, then no further messages need to be sent. However, if t_j *was* in $\tilde{T}_i$ after ODP this means that t_j could have been x_i's optimal state, so x_i must re-add any previously removed states to $\tilde{T}_i$, and re-send `ADD` messages in case the removal of t_j has impacted which states are dominated. As mentioned earlier, we must also consider scenarios with unreliable communication links. In these cases, as in FMS, both ODP and BnB FMS (explained next) messages will be sent and re-sent at regular intervals to ensure that the most up-to-date information is used in the decision making at each agent. After ODP completes, the operation of BnB FMS continues.

4.3.1.2 Branch-and-bound

Once ODP has completed, if, for any x_i, $|\tilde{T}_i| > 1$ (i.e., pruning was unable to solve the problem completely), then x_i will begin the execution of FMS by sending q messages as described in Equation 4.2. However, if $|\tilde{T}_i| = 1$, then FMS does not need to be run at x_i, as the domain has been pruned to the solution. If FMS does need to be run, then the agent proceeds with FMS when it receives r messages (see Equation 4.3) from its neighbours.

The final element of our algorithm is the use of branch-and-bound search in FMS: specifically, when a function node is performing its maximisation in order to reduce the joint variable state space that needs to be explored. Now, as given earlier, in Equation

4.3, the maximisation at a factor is a computational bottleneck, because a function node f_j must enumerate all $2^{|\mathcal{T}_j|}$ valid states and choose the one with the maximum utility. BnB MS involves a similar technique to this, but does not exploit the fact that $V(\emptyset, t_j) = 0, \forall t_j \in \mathcal{T}$, instead considering every possible state a variable can take. In contrast, we specialise this to task allocation and reduce the size of the search tree by only considering whether or not an agent is assigned to a task.

In this phase, we use search trees to prune joint states from the state space being explored at each function in the factor graph. To reduce this state space, we build a search tree, where each node is a partial joint state $\hat{\mathbf{x}}$ (where some variable states are not yet set), and leaf nodes are full joint states (where the values of all variables are set). We apply branch-and-bound on this search tree by computing the maximum value that can be gained from each branch (combining both utility and received messages), and pruning dominated branches. To do this, we first rename the expression in brackets from Equation 4.3 as $\tilde{r}$, so that message r is:

$$r_{j\to i}(I_{ij}(x_i)) = \max_{\mathbf{x}_j \setminus x_i} \tilde{r}_{j\to i}(\mathbf{x}_j) \tag{4.7}$$

where:

$$\tilde{r}_{j\to i}(\mathbf{x}_j) = f_j(\mathbf{x}_j) + \sum_{x_k \in X_j \setminus x_i} q_{k\to j}(I_{kj}(x_k)) \tag{4.8}$$

Hence, we wish to find $\max_{\mathbf{x}_j \setminus x_i} \tilde{r}_{j\to i}(\mathbf{x}_j)$.

We give our online branch and bound algorithm in Algorithm 4.5 (specifically, procedure `findMax`), and explain it below. First, we construct our search tree (line 2): rooted at $\hat{\mathbf{x}}_r = \{-, \ldots -, x_i, -, \ldots, -\}$, where $-$ indicates an unassigned variable, x_i is the variable that the message is being sent to, that is, the state that is fixed for that message. Then, in line 6, we choose the first unassigned variable in $\hat{\mathbf{x}}_r$ (say, x_1), and create a branch of the tree for each possible value of that variable — in our formulation, this would form two more partial states, $\{(x_1 = t_j), -, \ldots -, x_i, -, \ldots, -\}$ and $\{(x_1 \neq t_j), -, \ldots -, x_i, -, \ldots, -\}$ (line 7). Next, we estimate upper and lower bounds on the maximum value of $\tilde{r}_{j\to i}(\hat{\mathbf{x}})$ that can be obtained by further completing the partial state for each subtree (line 8). In more detail, the upper bound is computed as follows:

$$\tilde{r}_{j\to i}(\hat{\mathbf{x}})^{ub} = f_j(\hat{\mathbf{x}})^{ub} + \tilde{q}(\hat{\mathbf{x}})^{ub} \tag{4.9}$$

where $f_j(\hat{\mathbf{x}})^{ub}$ is found, depending on the function, by brute force or otherwise, and $\tilde{q}(\hat{\mathbf{x}})^{ub}$ is computed as:

$$\tilde{q}(\hat{\mathbf{x}})^{ub} = \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus i, \\ \hat{x}_k \neq -}} q_{k\to j}(I_{kj}(\hat{x}_k)) + \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus x_i, \\ \hat{x}_k = -}} \max_{x_k \in t_j, t_{-j}} q_{k\to j}(I_{kj}(x_k)) \tag{4.10}$$

where t_{-j} indicates any value other than t_j. Similarly, we compute the lower bound as

Algorithm 4.5 Online Joint State Pruning at function f_j

Procedure `findMax`(x_i)
$\hat{\mathbf{x}}_r = \{-, \ldots -, x_i, -, \ldots, -\}$
Return `expandTree`($\hat{\mathbf{x}}_r$)
Procedure `expandTree`($\hat{\mathbf{x}}$)
if $\hat{\mathbf{x}}$ is not fully assigned **then**
  x_k = first unassigned variable in $\hat{\mathbf{x}}$
  $\hat{\mathbf{x}}_{true} = \hat{\mathbf{x}} \cup (x_k = t_j)$; $\hat{\mathbf{x}}_{false} = \hat{\mathbf{x}} \cup (x_k \neq t_j)$
  Compute $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{true})^{ub}$, $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{true})^{lb}$, $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{false})^{ub}$ and $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{false})^{lb}$
  if $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{true})^{ub} < \tilde{r}_{j\to i}(\hat{\mathbf{x}}_{false})^{lb}$ **then**
    Return `expandTree`($\hat{\mathbf{x}}_{false}$)
  else if $\tilde{r}_{j\to i}(\hat{\mathbf{x}}_{false})^{ub} < \tilde{r}_{j\to i}(\hat{\mathbf{x}}_{true})^{lb}$ **then**
    Return `expandTree`($\hat{\mathbf{x}}_{true}$)
  end if
else
  Return $\tilde{r}_{j\to i}(\hat{\mathbf{x}})$
end if

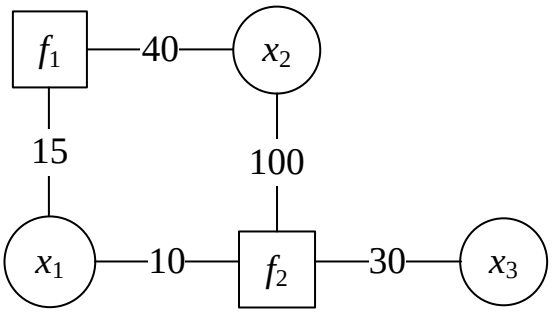


FIGURE 4.6: Demonstration environment representing a problem with 2 tasks and 3 agents. Circles are variables, squares are functions, and numbers on lines denote the utility of an agent being assigned to a task.

follows:

$$\tilde{r}_{j\to i}(\hat{\mathbf{x}})^{lb} = f_j(\hat{\mathbf{x}})^{ub} + \tilde{q}(\hat{\mathbf{x}})^{lb} \tag{4.11}$$

where $f_j(\hat{\mathbf{x}})^{ub}$ is, again, either found by brute force or otherwise, and $\tilde{q}(\hat{\mathbf{x}})^{lb}$ is computed as:

$$\tilde{q}(\hat{\mathbf{x}})^{lb} = \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus i, \\ \hat{x}_k \neq -}} q_{k\to j}(I_{kj}(\hat{x}_k)) + \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus x_i, \\ \hat{x}_k = -}} \min_{x_k \in t_j, t_{-j}} q_{k\to j}(I_{kj}(x_k)) \tag{4.12}$$

With these bounds, a branch can be pruned if the upper bound for $\tilde{\mathbf{x}}_{true}$ is lower than the lower bound for $\tilde{\mathbf{x}}_{false}$, or if the upper bound for $\tilde{\mathbf{x}}_{false}$ is lower than the lower bound for $\tilde{\mathbf{x}}_{true}$ (represented in Algorithm 4.5 lines 11–14 by not expanding pruned branches). The algorithm proceeds by expanding the next unassigned variable in the partial joint state represented by the remaining branch of the tree (line 6), and computing lower and upper bounds on the children. If possible, branches will be pruned, and so on, until we arrive at the leaves of the tree (where upper and lower bounds will be equal), which represent complete joint variable states (line 13), and we have found the joint state with the maximum utility (line 14).

Once a factor has run procedure `findMax` and found a search tree, the search tree is stored to reduce future computation. Then, when the factor receives a message from a variable in future, it uses the new message to recompute the bounds on the branches in the stored search tree. If the search tree is still valid (i.e., if pruned branches should still be pruned), then no computation needs to be done, and the function doesn't need to send any messages. However, if the bounds have changed enough that a previously-pruned branch needs to be expanded, then this branch must be expanded, and the algorithm will continue as it did before.

We prove that this phase of BnB FMS will not prune a locally optimal solution in Section 4.3.3. Given this, in the next section, we present a worked example of the operation of BnB FMS.

4.3.2 Worked Example

Here, we provide an example of our approach running in the simple environment given in Figure 4.6, where all functions are additive. Thus, the number on each edge in the figure denotes the utility of a given agent performing a given task, and the utility gained at a function from a certain set of agents completing it is the sum of these individual utilities.

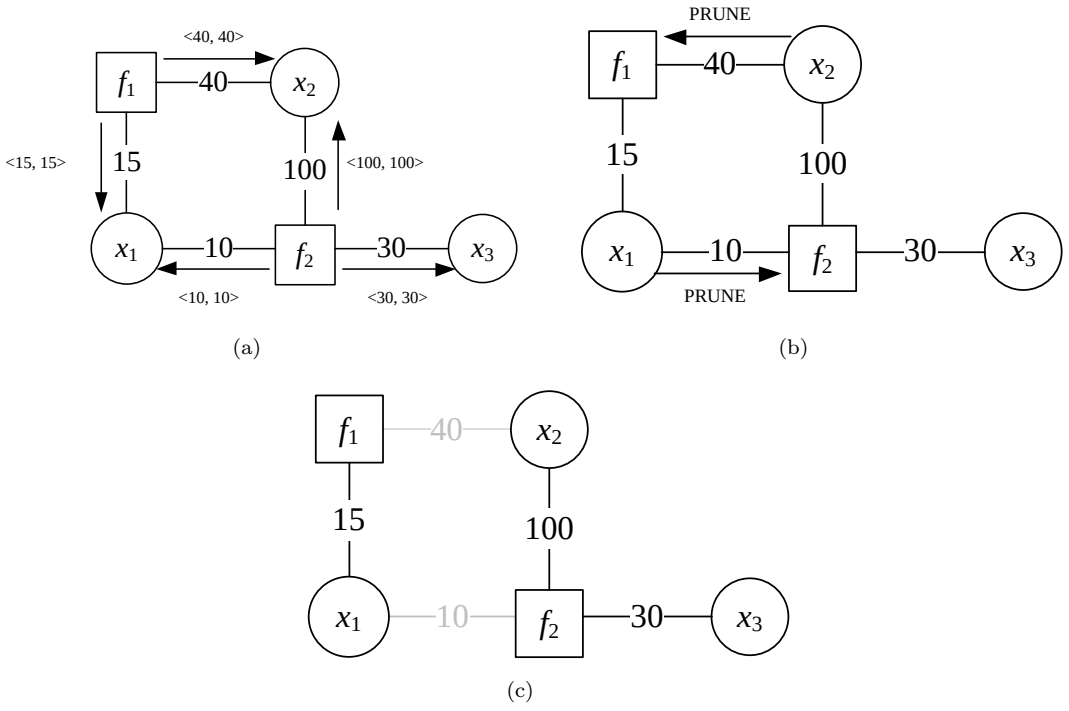


FIGURE 4.7: Example execution of online domain pruning. Circles are variables, squares are functions, and numbers on lines denote the utility of an agent being assigned to a task. Grey edges denote pruned domain elements, arrows represent messages being sent between functions and variables, and labels on arrows denote the contents of the messages.

In the first step of the algorithm, all functions send their upper and lower bounds to each variable, as shown in Figure 4.7(a). For example, function f_2 will send:

- **To x_1:** $\delta_2(x_1 = t_2)^{ub} = \delta_2(x_1 = t_2)^{lb} = 10$ — since functions in this example are additive (i.e., the utility gained at a function is the sum of the contributions of the agents assigned to its task), exact contributions are easily computed, and so the upper and lower bounds will be equal.

- **To x_2:** $\delta_2(x_2 = t_2)^{ub} = \delta_2(x_2 = t_2)^{lb} = 100$.

- **To x_3:** $\delta_2(x_3 = t_2)^{ub} = \delta_2(x_3 = t_2)^{lb} = 30$.

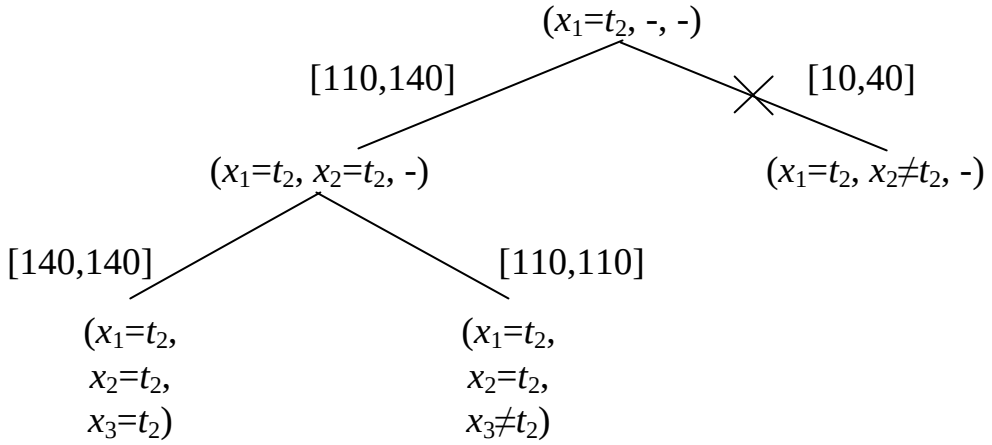


FIGURE 4.8: Example search tree at f_2.

As mentioned above, the functions here are additive, so messages sent from f_1 contain the number on the edge in Figure 4.7(a) as both upper and lower bounds.

Now, the receipt of these messages allows each variable node to decide which (if any) of its domain elements can be pruned. In more detail, domain element $t_j \in \tilde{T}_i$ will be pruned if $\delta_j(x_i)^{ub} < \max_{t_k \in \tilde{T}_i} \delta_k(x_i)^{lb}$ holds. In Figure 4.7, the decisions made at each variable node are as follows:

- At x_1: t_2 is pruned since $10 < 15$.
- At x_2: t_1 is pruned since $40 < 100$.
- At x_3: No domain elements are pruned since x_3 has domain of size 1.

Variable nodes then send their PRUNE messages to inform functions that they are no longer parameterised by those variables, as shown in Figure 4.7(b). The resulting graph with edges pruned is shown in Figure 4.7(c), which shows that each variable has domain of size 1, and so, in this case, the FMS step of BnB FMS wouldn't need to be run, and the algorithm would terminate here.

Nevertheless, for simplicity, in Figure 4.8, we show the search tree that would be explored at f_2, in order to send the message $q_{2\to1}(1)$, if no edges had been pruned by ODP. Here, we see that the $(x_1 = t_2, x_2 \neq t_2)$ branch of the tree would be pruned, since both its upper and lower bound are lower than the lower bound of $(x_1 = t_2, x_2 = t_2)$. In this case, it is clear that the state where all agents do t_2 is the best state. However, after function nodes have received messages from the variable nodes connected to them in the factor graph, these messages will be included when computing the bounds for each branch, as explained in Section 4.3.1.2. Thus, this example has shown how both phases of BnB FMS reduce computation done by FMS without removing any optimal states — a property which we formally prove in the next section.

4.3.3 Properties of BnB FMS

To show that BnB FMS is correct, we must show that the ODP and branch-and-bound phases of the algorithm will not prune optimal states from an agent's domain, under the assumption that an agent's optimal state only changes if there is a change in the environment. Thus, first we show that ODP is correct:

Theorem 4.2. *ODP never prunes optimal states from $\tilde{T}_i$.*

Proof. Let t^* be the optimal state of x_i. By Equation 4.6, in order to prune t^* from $\tilde{T}_i$, there must exist at least one $t_j \in T_i \setminus t^*$ where the following holds:

$$\delta_j(x_i, \mathbf{x}_{-i} \cup (x_i = t_j)) \geq \delta^*(x_i, \mathbf{x}_{-i} \cup (x_i = t^*)), \forall \mathbf{x}_{-i} \in \mathbf{X}$$

However, by definition, t^* is x_i's optimal state if and only if the following holds for all $t_k \in T_i \setminus t^*$:

$$\delta_k(x_i, \mathbf{x}_{-i} \cup (x_i = t_k)) \leq \delta^*(x_i, \mathbf{x}_{-i} \cup (x_i = t^*)), \forall \mathbf{x}_{-i} \in \mathbf{X}$$

Therefore, it is impossible for such a t_j to exist, and t^* can never be pruned. □

Therefore, since ODP cannot prune an optimal state from $\tilde{\mathcal{T}}$, it follows that ODP can never prune all states from $\tilde{\mathcal{T}}$. Given this, ODP will always converge to a point where no more states can be pruned, and so, ODP is correct. Next, we prove that the branch-and-bound phase of BnB FMS will not prune a locally optimal solution:

Theorem 4.3. *A joint state which is locally optimal at a function will not be removed from the search tree under Algorithm 4.5.*

Proof. $\delta_j(x_i \neq t_j, \mathbf{x}_j) = 0, \forall \mathbf{x}_j$. Therefore, $f_j(\mathbf{x}_j \cup (x_i = t_k)) = f_j(\mathbf{x}_j \cup (x_i = t_l)), \forall x_i \in \mathcal{X}_j, k \neq l$ and $k, l \neq j$. Hence, these two values are representative of the entire domains of $x_i \in \mathcal{X}$, and so all possible states of $x_i \in \mathcal{X}_j$ are covered by the search tree. Given this and the fact that the bounds used are correct, we can guarantee that the optimal joint state will not be pruned from the search tree. □

Therefore, under BnB FMS, the joint state chosen at each agent in response to the messages it has received will always be locally optimal, and so, phase 2 of BnB FMS is also correct.

Next, we show that BnB FMS is superstabilizing (see Section 2.2.5.1) subject to the same predicates given for FMS in Section 4.2.5 — specifically: *legitimacy*: $U(\mathbf{x}) = \max_{\mathbf{x}} \sum_{t_j \in \mathcal{T}} f_j(\mathbf{x}_j$, on acyclic graphs, where $U(\mathbf{x})$ is the total utility of assignment $\mathbf{x}$, and *passage*: the previous valid assignment of variables is maintained until a new solution has been computed.

Proposition 4.4. *BnB FMS is superstabilizing on tree structured graphs, with respect to the legitimacy and passage predicates given above.*

Proof. First, Theorems 4.2 and 4.3 are sufficient to show that the elements we have added to FMS satisfy the legitimacy predicate, since they do not remove optimal states. This, combined with Proposition 4.1, ensures that both predicates hold at all times. Hence, BnB FMS is superstabilizing. □

In addition to this, as with FMS, BnB FMS is anytime (see Section 2.2.5.2), since it can be stopped at any time during its execution and a solution can be computed.

4.3.4 Summary

Now, we have shown that BnB FMS addresses the scalability of FMS, at the expense of additional preprocessing and storage. Thus, BnB FMS should be used where scalability is important, and storage is cheap, and FMS should be used in smaller-scale environments. Additionally, we are still as yet only able to give quality guarantees on the solutions it finds on acyclic graphs (trees). In more detail, we can guarantee that FMS and BnB FMS will converge to an optimal solution on tree-structured graphs, but limited theoretical results exist as to the convergence of GDL algorithms (which FMS and BnB FMS are) on cyclic graphs. Thus, in the next section, we present BFMS, an algorithm to address the shortcomings of FMS and BnB FMS with regard to Requirement 6 (Boundedness).

4.4 Bounding the Solution Quality of FMS

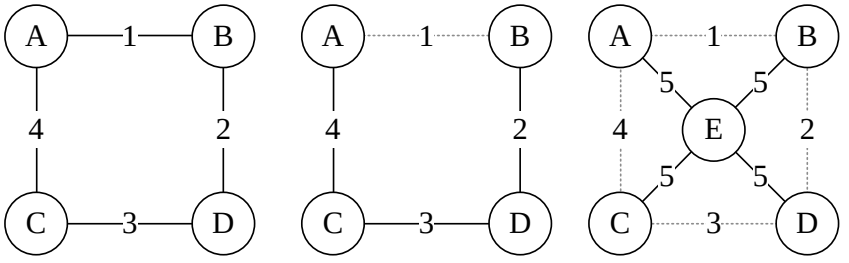
In this section, we describe our BFMS algorithm, which uses principles from BMS (see Section 2.3.5) in order to provide bounded approximate solutions on graphs containing cycles, at the expense of preprocessing and storage. Thus, BFMS is preferable for use where quality guarantees are important and storage is cheap.

Now, we cannot simply replace the max-sum element of BMS with FMS in a scenario such as ours, because every time the graph changes, the MST computed in BMS (to remove all cycles and attempt to minimise the loss in solution quality guarantees) and the solution should be recalculated in order to try and achieve the tightest bound possible. This is time-consuming and uses extraneous communication and computation. Given this, we have modified the GHS algorithm (used by BMS to find the MST, see Section 2.3.4) to recompute smaller sections of the graph in order to reduce overheads, while still improving the graph. We call this modified algorithm iGHS.

In what follows, we first explain the rationale for our modifications to GHS, describe of our iGHS algorithm, give a worked example of its operation, and show that it is superstabilizing and anytime, and then present BFMS.

4.4.1 iGHS

It is always necessary for the MST to be recalculated after the addition of a variable or function or removal of a variable or function from the underlying graph. We give an example of this in Figure 4.9, which depicts an example graph (Figure 4.9(a)), the MST of that graph (Figure 4.9(b)), and the new MST of that graph after node E has been added (Figure 4.9(c)). As can be seen, if we add node E to the tree, the original MST edges for nodes A, B, C, and D in Figure 4.9(b) no longer form an MST, as the edges connected to node E are weighted higher. Similarly, if node E were then to be removed from the graph, the MST would have to be recalculated, and the MST would return to that of Figure 4.9(b).



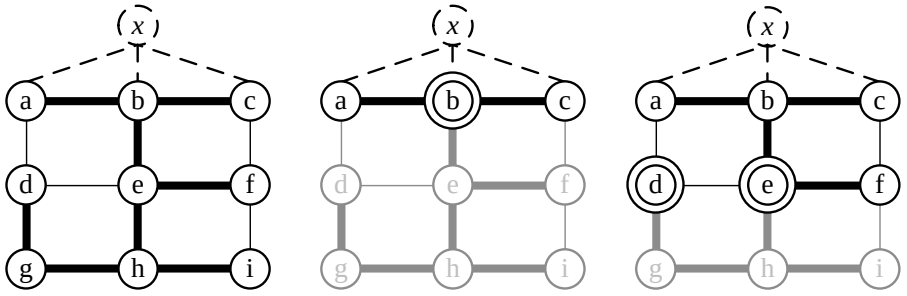
(a) Original Graph (b) MST of (a) (c) New MST after addition of node E.

FIGURE 4.9: Example scenario where adding a node changes the MST (numbers on edges denote edge weight, dotted edges denote edges in the graph but not the ST).

Despite the necessity of recalculating the tree upon a change in the graph in order for it to remain maximum, this is expensive in terms of communication and computation, particularly when re-computing over a very large graph. Additionally, the aim of finding an MST as opposed to an arbitrary ST as part of BFMS is to obtain a tighter bound on solution quality (see Section 4.4.2). Thus, at the expense of bound tightness, but to reduce communication and computation overheads, upon certain types of graph change, we only find the ST of a small part of the graph at a time. In so doing, we trade off communication and computation for the quality of the ST found, and so the tightness of the bound found by BFMS. Sacrificing bound tightness is permissible in many real-world dynamic task allocation problems, because it is preferable to get solutions quickly, as long as they are of acceptable quality. Later, we empirically show the tradeoff between communication and computation overheads and bound tightness (see Section 4.5). As such, we describe our iGHS algorithm in the remainder of this section.

4.4.1.1 The Algorithm

The general idea of iGHS is to run GHS (see Section 2.3.4) only on subgraphs of the whole problem. Specifically, given a ST, the whole factor graph, and a node to add, we run GHS on a subgraph $\mathcal{G}'$ of the graph $\mathcal{G}$ in order to find its MST. We choose $\mathcal{G}'$ by using a parameter k, which defines the depth of nodes in the graph we consider, measured from the node to be added. This is illustrated in Figure 4.10, where part (a) shows the original ST, with node x to be added, and parts (b) and (c) highlight the nodes which are included in subgraphs with $k = 1$, and $k = 2$, respectively.



(a) Original ST. (b) $k = 1$ (c) $k = 2$

FIGURE 4.10: The effect of k on the size of the subgraph. The dotted node and edges indicate a node to be added and the edges it introduces to the graph. Thick lines are ST branches, thin lines are edges in the graph but not the ST. Nodes with double outlines are frontier nodes, and faded nodes are nodes not in the ST.

Finding the MST of the subgraph while maintaining an MST over the rest of the graph is a challenge. In more detail, if two or more nodes in the subgraph have ST branches to nodes not in the subgraph (we call these *frontier nodes*: they are shown in Figure 4.10 as nodes with double outlines), then joining these two frontier nodes in the ST could cause a cycle in the ST as a whole, thus invalidating the tree. We can see how this would occur by considering nodes d and e in Figure 4.10(c): both nodes have a ST branch in the remainder of the graph, so making a ST branch either on edge (a, d) or edge (d, e) would connect d and e, thus creating a cycle in the rest of the graph. In order to combat this, we identify the frontier nodes, and ensure that we only connect to *one* of these nodes, on *one* edge. We can then guarantee we have not introduced a cycle.

Now, as we wish our algorithm to be completely decentralised, there can exist situations in which a node will need to process messages relating to two or more subgraphs at a time. To preserve the convergence property of the algorithm, we assume that each node n has a unique ID, $ID(n) \in \mathbb{Z}^+$. Thus, when we speak about *subgraph id*, we are referring to the subgraph created by iGHS for the node n with $ID(n) = id$.

iGHS proceeds in two phases: (1) a flooding phase which determines which nodes and edges are in subgraph *id*, and (2) a phase based on GHS, which establishes the MST on the subgraph, and adds the maximum weight frontier edge (i.e., edge joining to a frontier node) to the MST. We describe each of these phases next in turn.

Algorithm 4.6 Phase 1 of the iGHS algorithm, at a node n.

Require: $id, lastCount, possEdges, inEdges, exEdges = \emptyset$

At starting node, given k:
$lastCount_{id} = k$
$possEdges_{id} = adj(n)$
Send $Flood(k-1, id)$ to all $e_{ij} \in possEdges_{id}$
On receipt of $Flood(d, id)$ **on edge** e_{ij}**:**
if $lastCount_{id}$ doesn't exist **then** `// First message received about` id
  $lastCount_{id} = -\infty$
  $possEdges_{id} = adj(n)$
end if
if $d > lastCount_{id}$ **then**
  $lastCount_{id} = d; inEdges_{id} = inEdges_{id} \cup e_{ij}$;
  $possEdges_{id} = possEdges_{id} \setminus e_{ij}; exEdges_{id} = exEdges_{id} \setminus e_{ij}$;
  if $d < 0$ **then** `// Node` n `is not in the subgraph`
    Send $External(id)$ on edge e_{ij}, then **halt**.
  else `// Node` n `is in the subgraph`
    Send $Flood(d-1, id)$ on all $e' \in \{possEdges_{id} \setminus e_{ij}\}$
  end if
end if
if $possEdges_{id} = \emptyset$ **then**
  $lastCount = lastCount \setminus lastCount_{id}$
  Send $FloodAck(id)$ on edge e_{ij}, then **halt**.
else
  Put message on end of $queue_{id}$.
end if
On receipt of $External(id)$ **on edge** e_{ij}
$exEdges_{id} = exEdges_{id} \cup e_{ij}; possEdges_{id} = possEdges_{id} \setminus e_{ij}$
On receipt of $FloodAck(id)$ **on edge** e_{ij}
$inEdges_{id} = inEdges_{id} \cup e_{ij}; exEdges_{id} = exEdges_{id} \setminus e_{ij}; possEdges_{id} = possEdges_{id} \setminus e_{ij}$;

4.4.1.2 Phase 1: Flooding

Algorithm 4.6 gives the pseudocode for the flooding phase of our algorithm. In this phase, each node identifies which of its adjacent edges in the graph are within subgraph id ($inEdges_{id}$), and which are not ($exEdges_{id}$). This is done by initially putting all of a node's adjacent edges ($adj(n)$) into $possEdges_{id}$, which denotes that they could possibly belong to subgraph id.

In more detail, the flooding phase proceeds with the node to be added informing its neighbours of the value of k. More specifically, this node sends $Flood(k-1, id)$ messages along all of its unclassified neighbouring edges (line 3). Now, when a node n receives a $Flood(d, id)$ message, it will propagate further $Flood(d-1, id)$ messages along edges it is unsure of the status of (line 11), unless it receives a $Flood(d, id)$ message on edge e_{ij} containing a value less than 0. If this happens, the node sends an $External(id)$ message on edge e_{ij} (line 9), informing the node at the other end that e_{ij} is in $exEdges_{id}$ (line 15).

Now, in order for a node to classify an edge into $inEdges_{id}$, the node must have received either a $Flood(d, id)$ message (line 6) or a $FloodAck(id)$ message on that edge, for that id (line 16). $FloodAck(id)$ messages are sent when a node has classified all of its edges into $inEdges_{id}$ or $exEdges_{id}$ (line 12), and so, we can see that they are sent from the nodes that are furthest away, back toward the node with ID id. The algorithm stops when the changed node has received a $FloodAck(id)$ message from each of its neighbours. At that point, we know that every node in subgraph id is aware of which of its edges are in subgraph id, and which are not, and phase 1 of the algorithm is complete.

4.4.1.3 Phase 2: Modified GHS

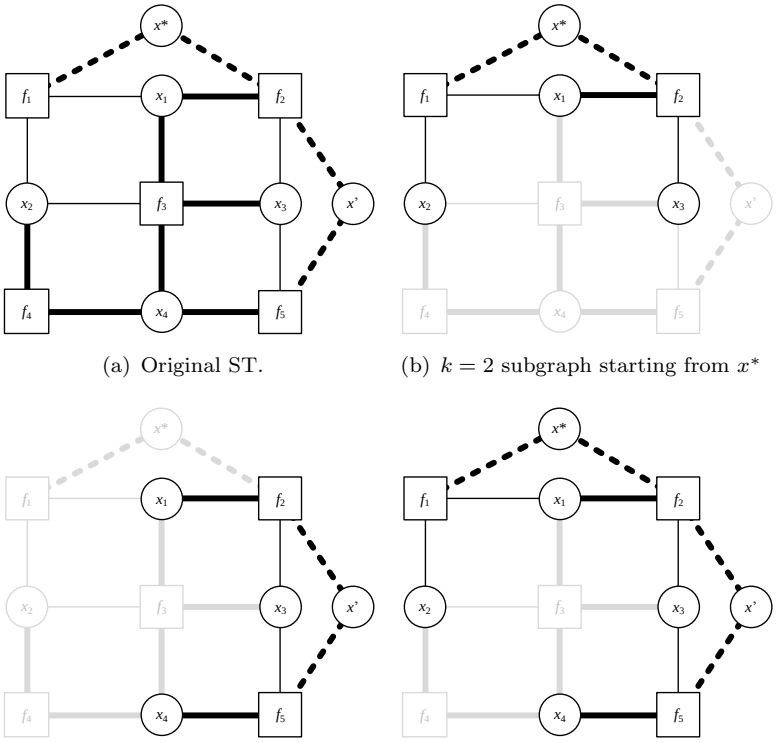
We give the pseudocode for phase 2 of iGHS in Algorithm 4.7. The pseudocode for sections of the algorithm that are repeated from GHS (lines 3, 9, 14, 20 and 35) can be found in Appendix A. This pseudocode is not repeated here so we can focus more on our algorithm.

Now, we must classify nodes into frontier and non-frontier nodes, in order to identify the edges in the subgraph that we must decide between, and avoid inadvertently creating a cycle. If more than one subgraph was found in step 1 (i.e., if more than one change was made to the graph at once), we must first merge the subgraphs before finding frontier nodes, in case the subgraphs overlap (if the subgraphs do not overlap, then this phase will still run in the same way, just multiple disjoint subgraphs will be created simultaneously). In more detail, Figure 4.11, depicts a graph with two nodes to be added, x^* and x' (see Figure 4.11 (a)). Having found the two $k = 2$ subgraphs shown in Figure 4.11 (b) and (c), the subgraphs need to be merged to form the subgraph depicted

Algorithm 4.7 Phase 2 of the iGHS algorithm, at a node n.

Require: $inEdges, exEdges, frontier$

1: **Procedure** `wakeup()`
2: $bestFrontierEdge = nil; bestFrontierWeight = \infty$
3: **if** $frontier = false$ **then** Proceed as GHS `wakeup()`
4: **On receipt of** $Connect(lv)$ **or** $Test(lv, F)$ **on edge** e_{ij}
5: **if** $frontier = true$ **then** // e_{ij} `is a frontier edge.`
6: $SE(e_{ij}) = REJECTED$ // e_{ij} `is not a branch of the spanning tree`
7: Send $Frontier$ on e_{ij} `// inform other node that` n `is a frontier node`
8: **else** // e_{ij} `is an edge in the subgraph.`
9: Proceed as GHS receipt of $Connect(lv)$ or $Test(lv, F)$
10: **end if**
11: **On receipt of** $InstConnect(lv)$ **on edge** e_{ij}
12: $SE(e_{ij}) = BRANCH$ // e_{ij} `is a branch of the spanning tree`
13: **Procedure** `test()`
14: **if** $\exists e_{ij}$s.t.$SE(e_{ij}) = BASIC \wedge e_{ij} \notin exEdges, frontierEdges\} \neq \emptyset$ **then** // `There are unclassified BASIC edges`
15: Proceed as GHS `test()` procedure.
16: **end if**
17: **On receipt of** $Frontier$ **on edge** e_{ij}
18: $SE(e_{ij}) = REJECTED$; // e_{ij} `is not a branch of the spanning tree`
19: $frontierEdges = frontierEdges \cup e_{ij}$ // e_{ij} `is a frontier edge`
20: $bestFrontierEdge = \min_{e' \in frontierEdges} weight(e')$ `// update best fronteir edge`
21: $bestFrontierWt = weight(bestFrontierEdge)$ `// update best frontier edge weight`
22: **if** $FN =$ nil **then** Proceed as last received GHS $Connect$
23: **else** `test()`
24: **Procedure** `report()`
25: **if** $findCount = 0$ and $testEdge = nil$ **then**
26: $SN = FOUND$ `// I have found my best edge`
27: Send $Report(bestWt, bestFrontierWt)$ on $inBranch$
28: **end if**
29: **On receipt of** $Report(wt, fw)$ **on edge** e_{ij}
30: **if** $e_{ij} \neq inBranch$ **then** `// If not received from tree parent`
31: $findCount = findCount - 1$ `// Decrement number of branches to test`
32: **if** $wt < bestWt$ **then** $bestWt = wt; bestEdge = e_{ij}$;
33: **if** $fw < bestFrontierWt$ **then** $bestFrontierWt = fw; bestFrontierEdge = e_{ij}$;
34: `report()`
35: **else if** $fw > bestFrontierWt$ and $wt = \infty$ **then** `// This is a frontier report message, not a normal GHS report message`
36: `ChangeFrontierRoot()`
37: **else** `// This is a GHS report message`
38: Proceed as GHS $Report(wt)$
39: **end if**
40: `ChangeFrontierRoot()`
41: **if** $SE(bestFrontierEdge) = BRANCH$ **then** `// Pass message on`
42: Send $ChangeFrontierRoot$ on $bestFrontierEdge$
43: **else** `// Make best frontier edge a spanning tree branch`
44: $SE(bestFrontierEdge) = BRANCH$
45: Send $InstConnect(lv)$ on $bestFrontierEdge$
46: **end if**



(a) Original ST. (b) $k = 2$ subgraph starting from x^*

(c) $k = 2$ subgraph starting from x' (d) The two subgraphs, combined

FIGURE 4.11: Example of how overlapping subgraphs are combined. Dotted edges indicate edges to be added to the graph when adding x^* and x'.

in Figure 4.11(d). To do this, at each agent, we set $inEdges = \bigcup_i inEdges_i$ (the set of all edges in all previously-found subgraphs), and $exEdges = \{\bigcup_i exEdges_i\} \setminus inEdges$ (the set of all external edges that are not within any other subgraphs). Then, if, for a node n, $exEdges \neq \emptyset$ and the previous status of at least one of the edges in $exEdges$ is $BRANCH$ (i.e., it was a branch in the ST pre-dating this computation), then n is a frontier node.

Next, the nodes must run iGHS in order to join into a tree and determine where the frontier nodes are in relation to the rest of the nodes. To do this, the nodes in the subgraph must first be woken up so that they can form into fragments (subgraphs). In more detail, when a node's status is $SLEEPING$, and it receives a message, it runs the `wakeup()` procedure (lines 1–3, Algorithm 4.7), in which the node will initialise its variables and try to connect to one of its neighbours (in order to form a fragment), if it is *not* a frontier node. When a node receives a connection attempt from its neighbour on edge e_{ij} (line 4), the algorithm will proceed in one of two ways: if the node is a frontier node, then it will mark e_{ij} as not in the ST and send a $Frontier$ message along e_{ij} (lines 5–7), and if the node is not a frontier node, then it will proceed as it would in GHS (line 8). When a node receives a $Frontier$ message along e_{ij} (line 15), it can mark e_{ij} as not in the ST (line 16), add e_{ij} to its list of frontier edges (line 17), and update its best frontier edge and weight values (lines 18–19), before carrying on as if edge j did

not exist (lines 20 and 21). Having connected on their best edges, each node will then try to connect on each of its other edges, in order of weight (see lines 12–14).

Once this has completed, the nodes must ascertain which is the best weight frontier edge to connect to the rest of the ST on. This is begun when a node has no more unclassified edges left: it will run the procedure `report()` (line 22, Algorithm 4.7), and inform its parent in the fragment of the highest weight frontier edge it has (lines 23–25). When the parent receives this report (line 26), it will decide which of its neighbours has the best weight frontier edge, and informs its own parent (lines 27–31). However, if the report message is received by the root of the fragment (line 32), then the receiving node can be sure that its best frontier edge leads to the ST's best frontier edge. As such, the receiving node sends an instruction to connect along the best frontier edge (line 33). Finally, if a node receives an instruction to join on its best frontier edge (line 36), it determines whether its best frontier edge points further down the tree (line 37), or is the edge to connect on (line 39), and either passes the message on (line 38), or connects on the best frontier edge (lines 41, 10–11).

Once the algorithm has completed, the ST of the subgraph minus the frontier nodes is maximum, and is connected to the ST of the overall graph on the maximum value frontier edge. Next, we detail how we handle the removal of nodes.

4.4.1.4 Managing Disruptions

Now, while adding a node to the graph is always handled in the same manner (i.e., creating a subgraph from the new node and running iGHS), this is not the case for removing nodes. In more detail, we must consider both circumstances under which a (function or variable) node can be removed from the graph: first, physical disconnection, such as when a rescue agent is malfunctioning or disappears, and second, when a task has been completed:

1. **Physical disconnection** — In order to support physical disconnection, we introduce contingency plans. Each time a node n is certain of the status of each of its incident edges (i.e., when they have all been marked $BRANCH$ or $REJECTED$), it chooses its maximum weight, and hence, most important, $BRANCH$ edge and informs the node at the other end (n_2) that n_2 is n's contingency. Then, if node n is removed from the graph, n_2 will instigate the iGHS algorithm, thus creating an MST around n_2.

2. **Task completion** — This case is slightly different, in that the factor node for that task will not be physically removed from the graph. Instead, the factor node acts as a 'handler' for iGHS, by setting the weight of each of its incident edges to 0 (as their utility would be 0 anyway), and instigating execution of iGHS.

The ability of iGHS to respond to disruptions comes at the cost of additional storage (over GHS) at each node, which is as follows:

$$\mathcal{S}_i = 2|E_i| + 4 \tag{4.13}$$

where $\mathcal{S}_i$ is the total number of values stored at node i, E_i is the set of edges incident on node i (counted twice because of the set $OldSE$), and 4 represents a value for each of $lastCount$, $frontier$, $bestFrontierEdge$ and $bestFrontierWt$.

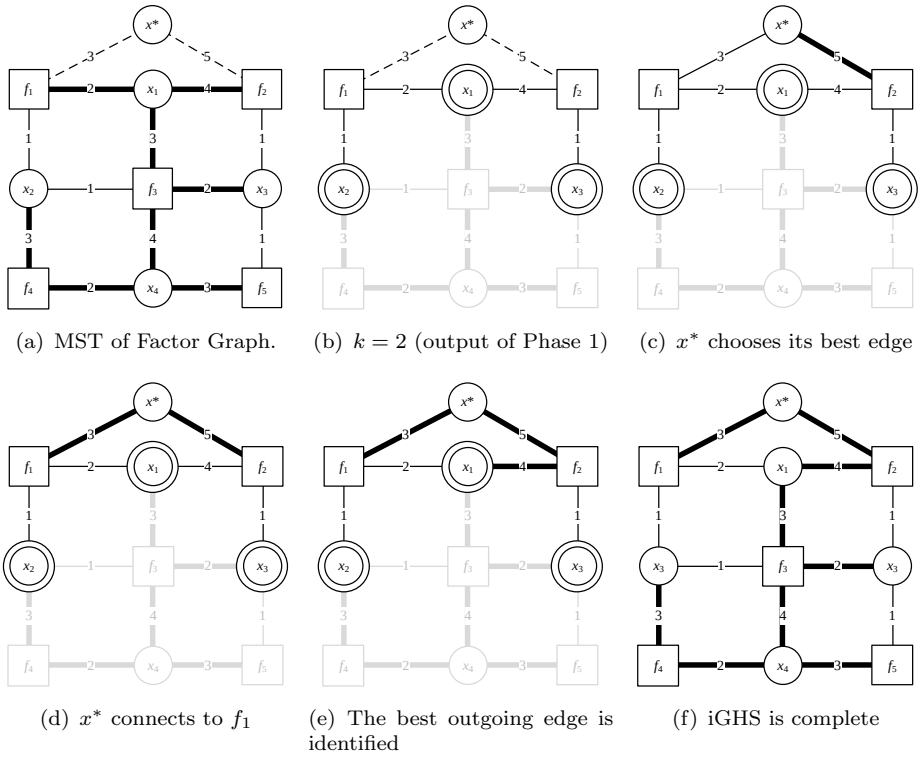
Nevertheless, in cases of unreliable communication links, graph nodes which are waiting for responses from other nodes will wait a predefined amount of time (which will depend on a number of scenario-specific parameters: for example, the size of the graph and the latency of communication links), before re-sending their messages, until they receive a response. We next provide a worked example of iGHS.

4.4.1.5 Worked Example

In this section, we provide a worked example of the operation of iGHS on a modified version of the example graph given in Figure 4.10. We have added edge weights to the graph, and removed an edge so that we can convert the graph into a factor graph (see Figure 4.12(a)).

Now, the first phase of iGHS is message flooding in order to ascertain which nodes and edges are in the $k = 2$ subgraph, shown in Figure 4.12(b). This is done by x^* sending a $Flood(1, *)$ message to its neighbours, f_1 and f_2 (line 1, Algorithm 4.6). On receipt of this, f_1 and f_2 send the $Flood(0, *)$ message to all of their neighbours, including x^* (line 4, Algorithm 4.6). As this message has a lower value of k than x^*'s own value, x^* ignores this message (line 5, Algorithm 4.6). x_1, x_2 and x_3 pass the $Flood(-1, *)$ message onto their neighbours — this is ignored by f_1 and f_2, as they know their depth is 0. f_3, f_4 and f_5 respond to these messages with $External(*)$ (line 9, Algorithm 4.6) — indicating they are outside of the subgraph being considered. As such, x_1, x_2 and x_3 have now classified all of their edges, and so, send $FloodAck(*)$ to f_1 and f_2 (line 14, Algorithm 4.6). On receipt of this (line 18, Algorithm 4.6), f_1 and f_2 have successfully classified all of their edges, so they, too, can respond to x^* with $FloodAck(*)$. Now that x^* has received $FloodAck(*)$ on both of its neighbouring edges, phase 1 of the algorithm is complete.

Now, to begin phase 2 of the algorithm, x^* calls the `wakeup()` procedure (line 1, Algorithm 4.7) and tries to connect on its highest value edge, $x^* \rightarrow f_2$ (Figure 4.12(c)). On receipt of this message, f_2 will send $Connect$ messages to x_1, and then x_3, but each will reply with $Frontier$ messages (line 7, Algorithm 4.7), allowing f_2 to classify edges leading to them as frontier edges (line 17, Algorithm 4.7). Hence, as f_2 has now classified all of its edges, it eventually calls the `report()` method (line 24, Algorithm 4.7), and



(a) MST of Factor Graph. (b) $k = 2$ (output of Phase 1) (c) x^* chooses its best edge

(d) x^* connects to f_1 (e) The best outgoing edge is identified (f) iGHS is complete

FIGURE 4.12: Example execution of iGHS on a factor graph when node x^* is added. As before, thick lines are ST branches, thin lines are edges in the graph but not the ST, double-outlined nodes are frontier nodes, and numbers on edges represent edge weights.

informs x^* that its best frontier edge is $f_2 \rightarrow x_1$, with a value of 4 (line 27, Algorithm 4.7). Having received this, x^* runs `test()` (line 13, Algorithm 4.7) and sends a *Test* message to f_1, which receives the message, calls `wakeup()` and sends *Connect* to x^* (its highest weight edge). x^* connects to f_1, as shown in Figure 4.12(c). Having done this, f_1 tests its other edges, discovering that x_1 and x_2 are frontier nodes. Hence, f_1 sends a *Report* message to x^* informing it that its best frontier edge is $f_1 \rightarrow x_1$, with a value of 2.

x^* can now calculate that the best frontier edge can be reached through f_2, and sends a *ChangeRoot* message to f_2 (line 36, Algorithm 4.7). f_2 can then send an *InstConnect* message to x_1 (line 45, Algorithm 4.7), informing it to make the connection without calling `wakeup()` and testing any other edges (line 12, Algorithm 4.7), as shown in Figure 4.12(e).

The algorithm is now complete, and we are left with the ST shown in Figure 4.12(f). Interestingly, in this example, the ST we are left with is the MST for the new graph — which means that we have achieved the same endpoint as running GHS would have, without the computation and communication overheads that GHS induces. However, this is unlikely to happen in general.

4.4.1.6 Properties of iGHS

Having presented our algorithm, we now show that it is correct, anytime (see Section 2.2.5.2), and superstabilizing (see Section 2.2.5.1).

Correctness. In order to prove the correctness of iGHS, we must show first, that it cannot create a cycle in the overall ST of the graph, and second, that it will make an MST in the subgraph. First, given that GHS is correct (Gallager et al., 1983), and cannot create cycles, and that iGHS operates by essentially running GHS on a portion of the subgraph, we know that iGHS will not make a cycle in that portion of the subgraph. Given this, it is sufficient for us to show that, in joining our MST to the ST of the rest of the graph, we cannot create a cycle. This is because if a node is connected to a tree-structured graph on a single edge, then it is impossible for that node to have created a cycle in the overall graph. Now, as we are sure that we make a ST across all edges but those that connect to frontier nodes, and we connect that tree to the rest of the graph on a single edge, we can guarantee that our algorithm can, indeed, not make a cycle in the graph overall. Second, we can guarantee that the ST produced by running iGHS on the subgraph is maximum due to the properties of the GHS algorithm.

Anytime. In order for our algorithm to be anytime, we ensure that for each edge e_{ij} incident on a node in the graph, we store two statuses, namely $SE(e_{ij})$ (the current state), and $OldSE(e_{ij})$ (the previous state). These variables are updated at each node once the iGHS algorithm has completed. To do this, we move the values from SE to $OldSE$. Thus, we ensure that the tree is maintained at all times, including when the iGHS algorithm is being run (before the values are changed over), and so, an ST can be generated at any time, and solutions produced will improve over time. iGHS is an anytime algorithm because it can always give a solution (using the values in $OldSE$), and will improve upon that solution over time (as nodes are added).

Superstabilization. In order to show that iGHS is superstabilizing, we define the following predicates: *legitimacy*: when the algorithm is complete, the ST produced contains no cycles, *passage*: at no time during recovery from a perturbation, are any cycles inadvertently formed. iGHS is superstabilizing with respect to these predicates: first, because iGHS does not form any cycles at any time in its operation (satisfying the passage predicate), and second, because iGHS is correct — and so, produces an acyclic graph at the end of its operation, satisfying the legitimacy predicate.

Now that we have ascertained these properties, we present BFMS, which combines iGHS and FMS in order to solve DCOPs on arbitrary graphs.

4.4.2 Bounded Fast-max-sum

Here, we introduce the BFMS algorithm, which consists of three main procedures running sequentially after a node is added to a graph:

1. **Efficient, superstabilizing ST generation**: using iGHS (Section 4.4.1) to recalculate a MST around the added node.

2. **FMS**, to calculate the optimal assignment of variables in the ST. Only those nodes whose individual utility changes as a result of the addition need to resend their FMS messages through the tree: if their individual utility does not change, then FMS will ensure messages are not transmitted unnecessarily (see Section 4.2.3).

3. **WSUM and SOLUTION propagation**, in order to bound the distance between the value of the solution obtained and the value of the optimal solution, using techniques from BMS (see Section 2.3.5).

An overview of these processes is shown in Algorithm 4.8, which indicates which Algorithms describe their operation in closer detail.

Algorithm 4.8 BFMS — overall algorithm execution after a change in the graph $\mathcal{FG}$.
Require: $\mathcal{FG}$, k
1: Weight all dependency links $e_{ij} \in \mathcal{E}$
2: Run iGHS using k: see Algorithms 4.6 and 4.7
3: Run FMS at variable and function nodes: see Algorithms 4.1 and 4.2
4: $WSUM$ and $SOLUTION$ propagation: see Algorithm 4.9

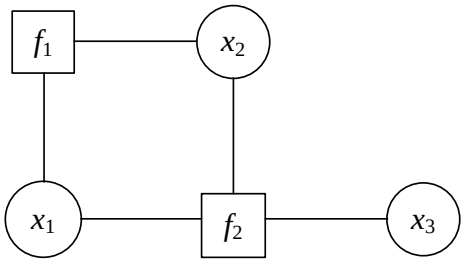


FIGURE 4.13: Example factor graph to demonstrate weight calculation.

As in BMS, each dependency link e_{ij} in the factor graph is weighted (line 1, Algorithm 4.8) as given below:

$$w_{ij} = \max_{\mathbf{x}_i \backslash j} \left[\max_{x_j} f_i(\mathbf{x}_i) - \min_{x_j} f_i(\mathbf{x}_i) \right] \tag{4.14}$$

For example, in Figure 4.13 the weight of the edge between x_2 and f_2 is computed as follows:

$$w_{22} = \max_{x_1, x_3} \left[\max_{x_2} f_2(x_1, x_2, x_3) - \min_{x_2} f_2(x_1, x_2, x_3) \right]$$

This weight represents the maximum impact that variable x_i can have over function f_j. Thus, by not including link e_{ij} in the ST, the distance between our solution and the optimal is at most w_{ij}.

Now, once each link has been weighted, iGHS is started on the factor graph (line 2, Algorithm 4.8 — see Section 4.4.1 for more details). The outcome of iGHS is, initially, an MST. Following this, after every addition to the graph, a subgraph around the addition is maximised, thus improving a section of the graph. As in BMS, each function f_i within the factor graph is required to keep track of any of its links that have been removed, denoted $\mathbf{x}_i^c$. We denote the set of links that have not been removed as $\mathbf{x}_i^t$, and thus, it follows that $\mathbf{x}_i = \mathbf{x}_i^c \cup \mathbf{x}_i^t$. Hence, as in BMS, we define the sum of removed weights as follows:

$$W = \sum_{e_{ij} \in C} w_{ij} \tag{4.15}$$

where C is the set of links removed from the factor graph. We also introduce the term W_{f_i}, which denotes the total weight of the edges removed from function f_i, or, more formally:

$$W_{f_i} = \sum_{x_j \in \mathbf{x}_i^c} w_{ij} \tag{4.16}$$

Next, we run FMS (line 3, Algorithm 4.8 — see Section 4.2) on the overall ST, beginning at the leaves, and propagating up the tree, with each node waiting to receive messages from all of its children before calculating and sending its next message. These messages then propagate up the tree in a similar manner — once they reach the root, the algorithm converges, and a variable assignment $\tilde{\mathbf{x}}$ can be computed. Functions with removed dependencies are evaluated by minimising over the removed dependencies, as given below:

$$\tilde{\mathbf{x}} = \arg\max_{\mathbf{x}} \sum_i \min_{\mathbf{x}_i^c} f_i(\mathbf{x}_i) \tag{4.17}$$

where $\tilde{\mathbf{x}}$ denotes the approximate solution: the solution gained through applying FMS to an ST of the original graph. Following this step, each node is aware of $\tilde{\mathbf{x}}$, and $\tilde{T}^m = \sum_i \min_{\mathbf{x}_i^c} f_i(\tilde{\mathbf{x}}_i)$, which will eventually be used to calculate the approximation ratio of the graph, $\rho(\mathcal{G})$ (given later in Equation 4.22).

Finally, the algorithm enters the WSUM and SOLUTION propagation phase (line 4, Algorithm 4.8, shown in more detail in Algorithm 4.9), which is the same as that of BMS. More specifically, once the leaves have received the FMS messages, they can compose new WSUM and SOLUTION messages (lines 1–3, Algorithm 4.9). These messages are composed by the leaf nodes and passed up the tree — once a node has received these messages from all of its child nodes, it will combine them and send a new message on to its parent (lines 4–8, Algorithm 4.9). Note that combination of these values for WSUM and SOLUTION messages is done differently at factor and variable nodes — we elaborate on this below.

Algorithm 4.9 $WSUM$ and $SOLUTION$ propagation at each node.

Require: *children*, *parent*

1: **if** $children = \emptyset$ **then**
2: Calculate $WSUM$ (Equations 4.18 and 4.19) and $SOLUTION$ (Equations 4.20 and 4.21)
3: Send $WSUM$ and $SOLUTION$ to *parent*
4: **else if** Received from node $n \in children$ **then**
5: **if** $WSUM$ and $SOLUTION$ received from all $n \in children$ **then**
6: **if** $parent \neq \emptyset$ **then**
7: Calculate $WSUM$ (Equations 4.18 and 4.19) and $SOLUTION$ (Equations 4.20 and 4.21)
8: Send $WSUM$ and $SOLUTION$ to *parent*
9: **else** `// Node is root of ST`
10: Calculate $WSUM$ (Equations 4.18 and 4.19) and $SOLUTION$ (Equations 4.20 and 4.21)
11: Send $WSUM$ and $SOLUTION$ to all $n \in children$
12: **end if**
13: **end if**
14: **else** `// Received from root of ST`
15: Forward $WSUM$ and $SOLUTION$ to all $n \in children$
16: **end if**

First, we detail the WSUM messages sent by the factor nodes and variable nodes. We begin by defining $WSUM_{f_j}$, the WSUM message sent by factor f_j to its parent node:

$$WSUM_{f_i} = W_{f_i} + \sum_{x_j \in children(f_i)} WSUM_{x_j} \tag{4.18}$$

where we use the term $children(f_i)$ to represent the set of f_i's child nodes. Thus, this message consists of the combined value of the weight of f_i's removed edges and the $WSUM$ messages received from f_i's children (if it has any). The WSUM messages sent by variable nodes differ slightly from this, as the removed edges are only represented at each function node (to avoid them being counted twice). Thus, we define the $WSUM_{x_j}$ message sent from x_j to its parent node as follows:

$$WSUM_{x_j} = \sum_{f_i \in children(x_j)} WSUM_{f_i} \tag{4.19}$$

where $children(x_j)$ represents the child nodes of x_j. From this, we can see that variable leaf nodes (i.e., nodes without children) will initially send an empty WSUM message.

Similar to the above, we denote the SOLUTION message sent from a function node f_i to its parent node as $SOLUTION_{f_i}$:

$$SOLUTION_{f_i} = f_i(\tilde{\mathbf{x}}_i) + \sum_{x_j \in children(f_i)} SOLUTION_{x_j} \tag{4.20}$$

Similar to the WSUM messages, the SOLUTION message from function node to its parent node consists of combining the solution value at function f_i with the SOLUTION messages received from its children. Finally, we define the message sent from a variable node x_j to its parent node as follows:

$$SOLUTION_{x_j} = \sum_{f_i \in children(x_j)} SOLUTION_{f_i} \tag{4.21}$$

Again, the SOLUTION message sent by a leaf variable node will be empty, as it has no child nodes to combine messages from. Given all this, once these messages reach the root, the root sums them and propagates them back down (lines 9–11, 14–15, Algorithm 4.9), so every node is aware of the total weight removed, W, and the solution value, $\tilde{\mathcal{T}} = \sum_{v \in \mathcal{T}} f_v(\tilde{\mathbf{x}}_v)$. Now the agents have all information to compute the approximation ratio, as follows:

$$\rho(\mathcal{G}) = 1 + (\tilde{\mathcal{T}}^m + W - \tilde{\mathcal{T}})/\tilde{\mathcal{T}} \tag{4.22}$$

where $\mathcal{G}$ is the factor graph the algorithm has been run on, $\tilde{\mathcal{T}}^m$ is the approximate solution value, and $\tilde{\mathcal{T}}$ is the actual solution value. In order to improve our approximation ratio, we wish to keep the value of W as low as possible, by only removing low-weight edges (i.e., edges that have little bearing on the overall solution). We can see that the

value of k given to iGHS has a bearing on the approximation ratio, too — higher values of k optimise larger sections of the graph. This leads to a larger portion of the resulting ST being maximum (and so, a lower W), and therefore means that approximation ratios will be closer to 1 for higher values of k. In addition to this, as we discuss later (in Section 4.4.4), k controls the competitive ratio achieved by BFMS in terms of the cost of changing variable assignments, too, since small values of k mean that only a small portion of the graph will be recalculated, and so, only a small portion of the variables are likely to change their assignments.

4.4.3 Worked Example

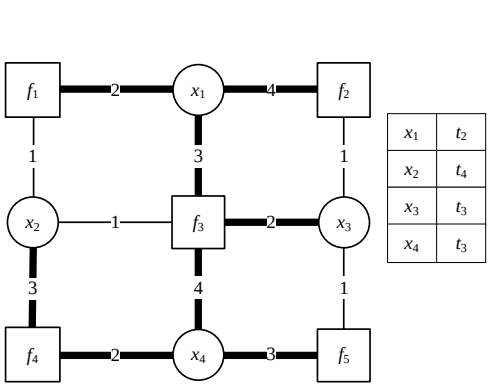
In this section, we give an example execution of BFMS on the graph given in Figure 4.14 (a). In the Figure, we give the task that each variable is assumed to be assigned to, and keep the edge weights we used in Section 4.4.1.5, for simplicity.

BFMS begins with the execution of iGHS, as explained in Section 4.4.1.5 — the outcome of this is shown in Figure 4.14 (b). Note that the assignments in Figures 4.14 (a) and (b) are identical — they do not change until the agent has made a new decision. This is so that the agents can continue working on their tasks after a change in the environment, until the algorithm is completed.

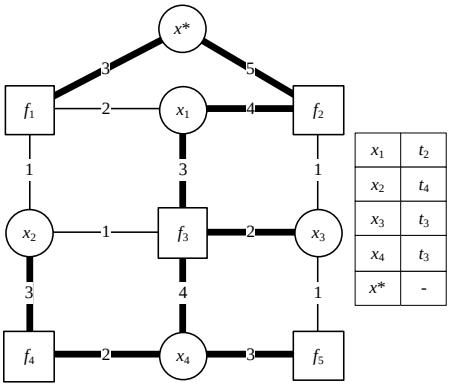
Now, the second phase of BFMS can begin: exchanging FMS messages in order for each agent to calculate its new variable value. Once x^* is aware that iGHS has completed, it sends its $q_{x^* \to f_1}$ and $q_{x^* \to f_2}$ messages to f_1 and f_2, as shown in Figure 4.14 (c). On receipt of this message, f_1 sends its FMS messages (see Equation 4.3) to its ST neighbours: i.e., just x^*, as shown in the left hand side of Figure 4.14 (d). f_2 is neighboured by x^* and x_1 in the ST, so it sends FMS messages (see Equation 4.3) to both nodes (again, as shown in Figure 4.14 (d)). On receipt of the messages from f_1 and f_2, x^* can now choose its variable assignment, and assigns its variable to f_2. The message x_1 has received from f_2 will not have changed from the initial round of FMS messages (and so will not differ from x_1's stored values for f_2), and so x_1 does not send any further messages. This completes the second phase of BFMS.

For the final phase of BFMS, the nodes must communicate the total weight of edges not in the ST (WSUM), and the total solution value (SOLUTION). To do this, f_1 and x_1 send their WSUM and SOLUTION values towards x^* (Figure 4.15 (a)), with x_1 representing the subtree below it. f_2 receives WSUM and SOLUTION from x_1, adds to the WSUM value, and passes them toward x^* in Figure 4.15 (b). Now that x^* has received all of the values, it totals them and propagates them back down through the tree, as shown in Figure 4.15 (c).

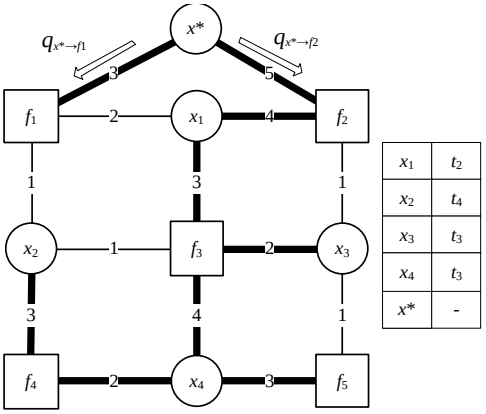
Given this, the approximation ratio can be calculated at any node in the factor graph, as given in Section 4.4.2. Note that, in our particular formulation, removing variable



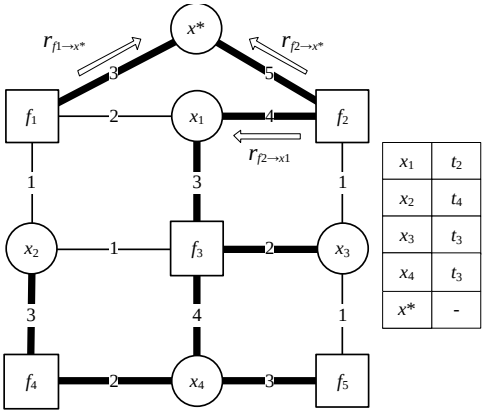
(a) Original graph and assignments.



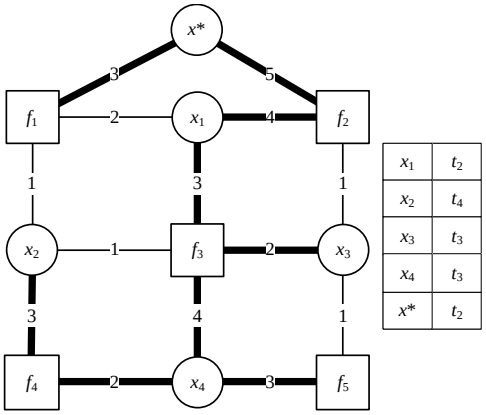
(b) ST from iGHS given the addition of x^*.



(c) x^* sends its messages

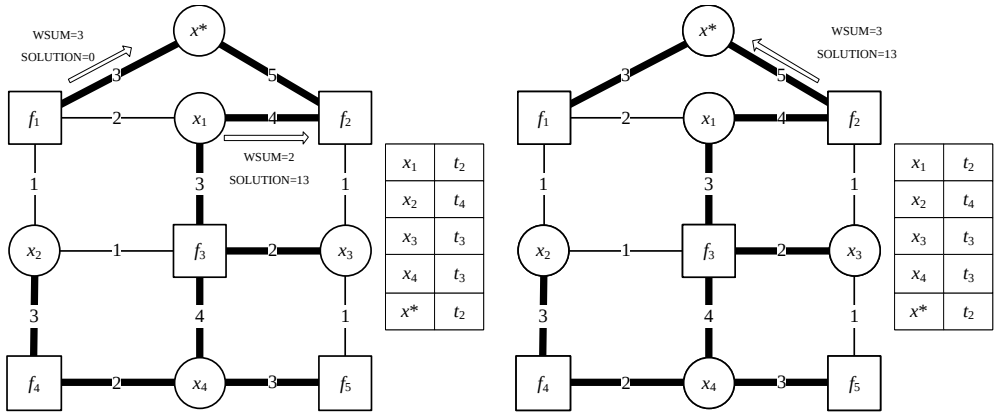


(d) f_1 and f_2 send their messages



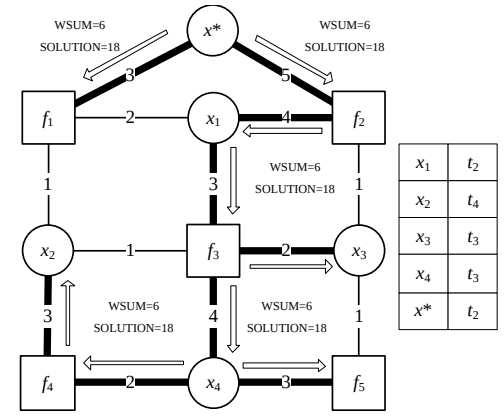
(e) x^* chooses assignment, x_1 sends no more messages

FIGURE 4.14: Example execution of parts 1 and 2 of BFMS on a factor graph. As before, thick lines are ST branches, thin lines are edges in the graph but not the ST, and numbers on edges represent edge weights.



(a) WSUM and SOLUTION from the leaves

(b) Continues to propagate



(c) Total WSUM and SOLUTION then propagate down

FIGURE 4.15: Example execution of part 3 of BFMS on a factor graph. As before, thick lines are ST branches, thin lines are edges in the graph but not the ST, and numbers on edges represent edge weights.

dependencies from functions also removes the tasks represented by those functions from the variables' domains. Thus, the solution gained by minimising over all removed dependencies is equal to the perceived actual solution value, as both are calculated over a variable's modified domain. Hence, we can simplify the approximation ratio equation to $1 + W/\tilde{T}^m$. Now we can calculate the approximation ratio of our example to be $1 + 6/18 = 1.33$.

Now, as we mentioned in our iGHS example in Section 4.12, in this case, the ST produced by iGHS is an MST. As such, the approximation ratio of 1.8 achieved by BFMS' combination of iGHS and FMS is exactly the same as the approximation ratio achieved by combining GHS with max-sum (as in BMS), only with communication and computation linear in the induced width of the graph, since we are applying a GDL algorithm to a (spanning) tree. We now go on to explain the key properties exhibited by BFMS.

4.4.4 Properties of BFMS

Here, we verify that BFMS is superstabilizing (Section 2.2.5.1) and anytime (Section 2.2.5.2). We first verify that BFMS is superstabilizing. We do this subject to the following predicates: *legitimacy*: $U(\mathbf{x}) = \max_{\mathbf{x}} \sum_{t_j \in \mathcal{T}} f_j(\mathbf{x}_j)$, on acyclic graphs where $U(\mathbf{x})$ is the total utility of assignment $\mathbf{x}$, and *passage*: the previous assignment of variables is maintained until a new solution has been computed. Given this, BFMS is superstabilizing, because FMS and iGHS (see Sections 4.2.5 and 4.4.1.6) are.

Now, BFMS is also anytime, because both iGHS and FMS maintain previous edge and variable states at all times, so a result can be obtained at any point in the execution of the algorithm. In addition, the iterative improvement aspect of iGHS means that solutions obtained will improve as changes are made to the graph.

In terms of competitive analysis, BFMS is more competitive than FMS, in terms of variable assignments, since, as we mentioned earlier, the value of k has a degree of control over the number of variables that change their values. In more detail, lower values of k are more likely to produce small subgraphs in iGHS, and so, only a small portion of the factor graph will change in the first phase of BFMS. This, then, means that fewer variable domains are likely to have changed, and so, fewer variables are likely to change their assignment for such low values of k. Conversely, as k tends to the width of the graph, the competitiveness of BFMS degrades to that of FMS, since more of the graph will be re-computed.

4.4.5 Summary

In sum, here we have presented our iterative MST algorithm, iGHS, and shown how it can be combined with FMS to find solutions with quality guarantees. This means that if a solution is required with quality guarantees, then BFMS should be used. However, if quality guarantees are not needed, the FMS or BnB FMS should be used. In the remainder of this section, we give our empirical evaluation, showing how FMS, BnB FMS and BFMS perform in practice, and how they meet the requirements that we defined in Section 1.3.

4.5 Empirical Evaluation

In this section, we evaluate the properties of FMS, BnB FMS and BFMS on different types of scenarios given that these algorithms are likely to perform differently given the density of the environment (i.e., the number of agents that can do each task, and the number of tasks that each agent can do), the structure of the environment (i.e., whether or not it contains cycles) and the scale (i.e., the number of agents and tasks) of

the environment. Density, structure and scale are the three main experiment variables, since they are the three things that have an impact on how difficult a DCOP is to solve.

To do this, we divide this section into three key experiments, evaluating the impact that density, structure and scale have on how well our algorithms meet the requirements set out in Section 1.3. Specifically, in experiment 1, we evaluate robustness to change (requirements 2, 3 and 4). In experiment 2, we evaluate scalability and quality of solutions and bounds found (requirements 1, 5 and 6). Finally, in experiment 3, we perform competitive analysis with respect to variable assignment changes on our algorithms (requirement 4). We describe our methodology, each of these experiments, and their results, in the remainder of this chapter.

4.5.1 Methodology

To evaluate our algorithms, we generated 200 random graphs, scale-free graphs and tree-structured graphs with varying link density. We evaluate FMS and BnB FMS on tree-structured graphs since they, and max-sum, which we compare them to in order to show the savings they provide, are only guaranteed to converge on tree-structured graphs (for an evaluation of the solution quality achieved by FMS on cyclic graphs, see Section 4.5.3). In contrast to this, BFMS and BMS were specifically created to converge on cyclic graphs, and so we evaluate BFMS on random graphs and scale-free graphs (which are generally accepted to be difficult graphs in the DCOP community), instead of trees. We generated these graphs as follows (where δ_x is the average degree of a variable, and δ_f is the average degree of a function, to simulate the graph sparsity which could come from agents which have limited capabilities or limited distances they can travel to tasks):

- **Random graphs:** To generate random graphs, we first build a spanning tree covering all of the agents and tasks. To do this, we choose a random task, and connect it to a random agent, and store this as our tree. Then, we choose a random task/agent, and connect that to a randomly-chosen agent/task which is already in the tree, and repeat this until every node is in the tree. Then, we randomly generate links between agents and tasks such that each agent is connected to an average of δ_x tasks.
- **Scale-free graphs:** To generate our scale-free graphs, we create $|\mathcal{A}|$ agents, and select the degree of each agent using a power law with power $\lambda = 2$.[4] We then randomly select a subset of δ_f agents with unassigned task slots, create a new task, and assign those agents to the task. We repeat this until all task slots are assigned. To choose a subset of agents, we select agents with unassigned task slots with an

[4]The power is usually set to 2 or 3 because values between 2 and 3 are widely accepted to be the most commonly found in real-life scale-free networks. For examples, see Albert and Barabàsi (2002).

independent probability $\frac{\delta_f}{N}$, where N is the number of agents with unassigned task slots to ensure that the tasks are connected to an average of δ_f agents.

- **Trees:** To generate trees with varying δ_x, we create one task and connect it to $U(3,6)$ agents. Then, we connect each of those agents to $\delta_x - 1$ tasks, which are then connected to $U(3,6)$ agents, which, finally, are connected to $\delta_x - 1$ tasks. This gives us a bipartite tree of depth 4, with each agent able to perform δ_x tasks. To generate trees with varying δ_f, we do the exact opposite of this.

In future work, it would be interesting to see how our algorithms perform on small world and lattice graph topologies, since small world graphs exhibit small groups of tightly connected nodes as could be found in a disaster management environment. We would expect our algorithms to do well on such graphs. In addition to this it might be interesting to get something approaching real-world data on the arrangement of agents and tasks in such environments. However, for the purpose of this thesis we focus on the three graphs listed above.

Given these graphs, in our robustness and competitive ratio experiments, we ran each algorithm on an initial graph, then performed n random changes[5] that graph, and ran the algorithm again. We repeated this 10 times for each graph, where n is drawn from a Poisson distribution for each timestep, with $\lambda \in \{1, 5, 10\}$ (i.e., 2%, 10% and 20% of the initial number of tasks), running each algorithm at each timestep. We used a random look-up table, drawn from a uniform distribution, as the utility function of each task in order to evaluate our algorithm in general task allocation domains.

Now, on the one hand, the degree of each factor node is equal to the number of variables it is connected to (i.e., how many agents can complete the task). Hence it determines the space of possible combinations (or coalitions) the factor needs to search through, which is exactly 2^{δ_f} combinations. On the other hand, the degree of each variable node is equal to the number of factors it is connected to, and hence the number of possible allocations (i.e., $\delta_x - 1$) it can affect if one of its factors is removed from the graph (i.e., a task is removed). Given these features of the problem, our goal is to evaluate how FMS, iGHS and BnB FMS can minimise the number of messages that need to be propagated in the graph and the computation performed by all remaining nodes when random changes are performed on the graph. In more detail, we recorded some typical measures used in the DCOP community (Modi et al., 2005):

Mean computation units used (MCU) — the average total number of combinations of states evaluated at all factor nodes.

[5]Chosen uniformly from adding or removing an agent or task, where existing tasks/agents are randomly selected and removed, and new tasks/agents are created with connections to δ_x randomly-selected tasks/agents, in order to maintain the chosen graph density.

Mean total number of messages sent (TNS) — the average total number of messages sent, by both variable and factor nodes.

Mean total size of messages sent (TSS) — the average total size of all messages sent by all nodes, measured in bytes. This reflects the number of total number of variable states communicated throughout the graph.

Mean total size of preprocessing messages sent (PTNS) —the average total size of all *preprocessing* (i.e., messages sent by iGHS/ODP) messages sent by all nodes, measured in bytes.

Against this background, in our robustness and competitive ratio experiments (experiments 1 and 3), we separately varied the values of δ_x and δ_f between 1 and 15, in increments of 1. As mentioned earlier, when generating trees, the variable that was not fixed (i.e. δ_f when we fixed δ_v, and vice versa) was randomly selected from a uniform distribution in [3, 6]. For each value of the fixed variable, 200 random tree/graph instances were generated.

For our solution quality experiments, we varied the total number of agents, number of tasks, and graph density of random and scale-free graphs, as follows:

- **Node degree:** $|\mathcal{A}| = 100$, $|\mathcal{T}| = 100$, and $\delta_x \in \{2, 3, 4, 5, 6\}$.
- **Number of agents:** $|\mathcal{A}| \in \{20, 40, \ldots, 200\}$, $|\mathcal{T}| = 100$, and $\delta_x = 3$.
- **Number of tasks:** $|\mathcal{A}| = 100$, $|\mathcal{T}| \in \{20, 40, \ldots, 200\}$, and $\delta_x = 3$.

Given all this, in the rest of this section, we present the results of our experiments.

4.5.2 Experiment 1: Robustness to Change

When disruptions occur in the environment, the current set of tasks and/or agents is increased or decreased and the agents may need to be re-assigned. In the worst case, this means that the whole allocation of agents to tasks needs to be recomputed. In this experiment, we evaluate the communication and computation used by our algorithms after a change in the graph, compared to naïve approaches which are simply re-run after a change in the environment.

To do this, we divide our three algorithms into two groupings: FMS and BnB FMS, and BFMS. We do this because FMS and BnB FMS were created to specifically address requirements 1, 2, 3, and 4 (scalability, robustness, efficient communication, adaptiveness), and BFMS was created to specifically address requirement 6 (boundedness). In addition to this, only BFMS has been proven to converge on graphs containing cycles

(FMS and BnB FMS have been proven to converge on acyclic graphs only), and so only BFMS will produce a deterministic solution on such graphs.

Given that we expect our algorithms to outperform their competitors on several fronts, we postulate a number of hypotheses:

1. FMS has lower MCU, TNS, and TSS for all values of δ_x and δ_f than max-sum. BnB FMS has lower MCU, TNS and TSS than FMS, for all values of δ_x. In addition, BnB FMS has lower PTNS than BnB MS.

2. BFMS has lower MCU and TSS than BMS, for all values of δ_x.

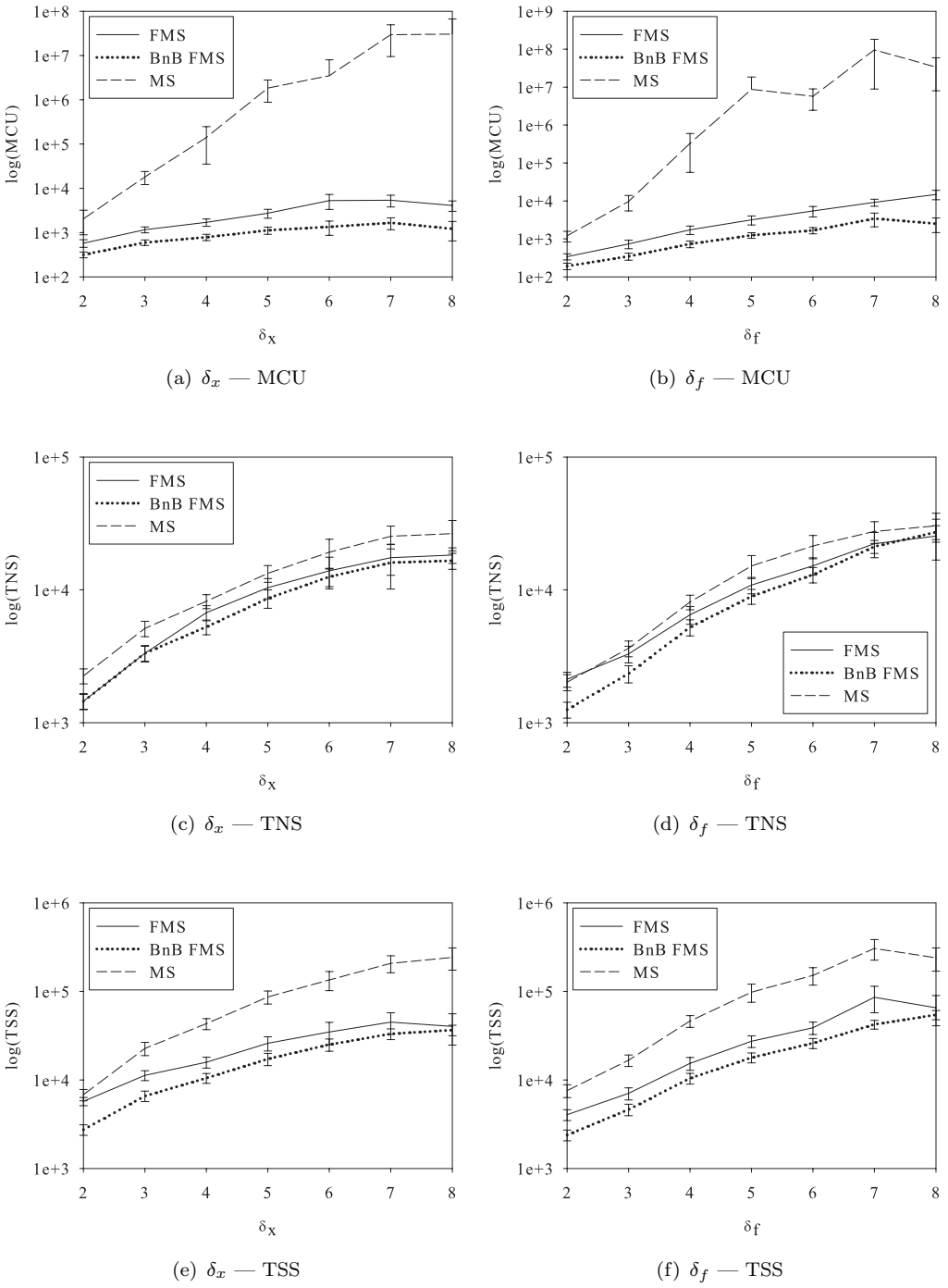
In what follows, we examine how our results perform against these hypotheses — we divide the discussion of results into two sections, one for comparing FMS, BnB FMS and max-sum, and one comparing BFMS and BMS.

4.5.2.1 Comparing FMS, BnB FMS and Max-sum

The results shown in Figure 4.16, plotted with 95% confidence intervals[6] as error bars to illustrate statistical significance, confirm most of hypothesis 1. We have compiled the best- and worst-case improvements of FMS and BnB FMS over max-sum in Table 4.2. An interesting outcome from these results is that FMS and BnB FMS are not significantly different in terms of the number of messages they send (i.e., the TNS measure) — whilst both clearly send fewer messages than max-sum, the difference here is not as large as we had expected, since changes to the graph made a difference to a greater number of variable values than we had expected. This goes to show that FMS and BnB FMS make the most significant savings in computation performed and in message *size* as opposed to message *amount*.

Our results show that, indeed, FMS and BnB FMS have the most significant improvements over max-sum in graphs where large numbers of factors are connected to each variable. However, for the lower values of δ_x, the difference in MCU and TSS of FMS, BnB FMS and max-sum is much smaller. This is because, when δ_x is low, the variables have a much smaller domain. In more detail, we showed in Section 4.2 that FMS reduces the MCU of max-sum from d^n to 2^n, and the size of each message sent to 2 in all cases, as opposed to it depending on d. Thus, it follows that FMS will only make a noticeable difference where $d > 2$. Given this, we can see that the improvement given by FMS increases very quickly as the value of δ_x grows. Additionally, we showed in Section 4.3 that BnB FMS not only reduces the domains of variables in the factor graph, but also

[6] 95% confidence intervals are computed as $1.96 \times SE$, where SE is the standard error, computed as $SE = \frac{s}{\sqrt{n}}$, where n is the sample size (in this case, 200), and s is the standard deviation in the mean, computed by $s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$, where $\bar{x}$ is the mean.



(a) δ_x — MCU

(b) δ_f — MCU

(c) δ_x — TNS

(d) δ_f — TNS

(e) δ_x — TSS

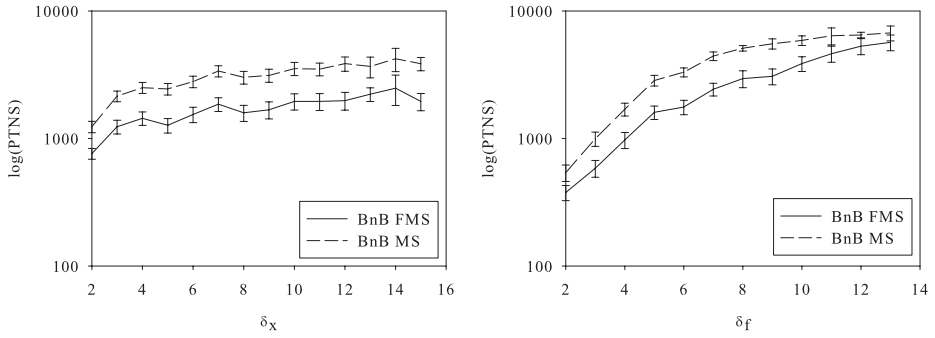
(f) δ_f — TSS

FIGURE 4.16: Comparing the robustness of FMS, BnB FMS, and max-sum (MS), where $\lambda = 5$, over increasing δ_x (edges per variable node) and δ_f (edges per function node).

	FMS			BnB FMS		
	MCU	TNS	TSS	MCU	TNS	TSS
$\lambda = \mathbf{1}$**:**						
δ_x	36–99%	36–47%	49–93%	66–99%	36–79%	49–96%
δ_f	71–99%	5–45%	67–81%	88–99%	18–58%	76–84%
$\lambda = \mathbf{5}$**:**						
δ_x	71–99%	18–35%	15–83%	84–99%	34–37%	59–84%
δ_f	71–99%	0–29%	46–74%	71–99%	10–41%	68–86%
$\lambda = \mathbf{10}$**:**						
δ_x	67–99%	0–21%	34–70%	80–99%	25–38%	63–77%
δ_f	69–99%	7–18%	48–68%	69–99%	31–39%	66-79%

TABLE 4.2: Minimum and maximum improvements over max-sum given by FMS and BnB FMS.

specifically reduces computation at factors as well. Thus, as expected, BnB FMS sends fewer messages and performs less computation than FMS. This demonstrates that FMS will always consider more states than BnB FMS, and so shows how BnB FMS advances towards requirements 1 and 3 (scalability and efficiency).



(a) δ_x (b) δ_f

FIGURE 4.17: Comparing the preprocessing messages (PTNS) used by BnB FMS and BnB MS, with $\lambda = 1$.

In addition to this, Figure 4.17 shows that the ODP phase of BnB FMS sends up to 49% fewer messages (when $\delta_x = 15$, and $\lambda = 1$ — performance approaches BnB MS with increasing λ) than the domain pruning element of BnB MS, as a result of specifically catering for dynamic environments, whilst achieving the same utility as FMS, and, as we showed earlier, achieving lower values of MCU, TNS, and TSS than FMS. In addition, the saving in messages is almost invariant in the graph density because the number of messages sent is linked to the number of states pruned from the variables' domains, as pruning states is equivalent to pruning edges in the factor graph.

As mentioned earlier, the computational and communicational savings made by BnB FMS come from the fact that domain elements are pruned by ODP in phase 1. Thus, in Figure 4.18, we demonstrate how the total number of domain elements pruned varies

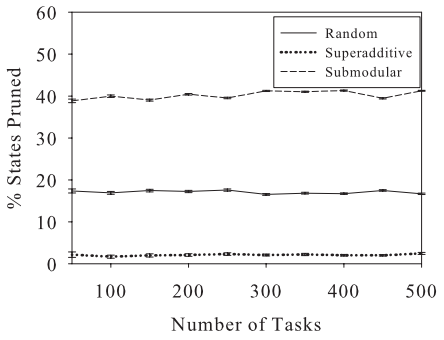


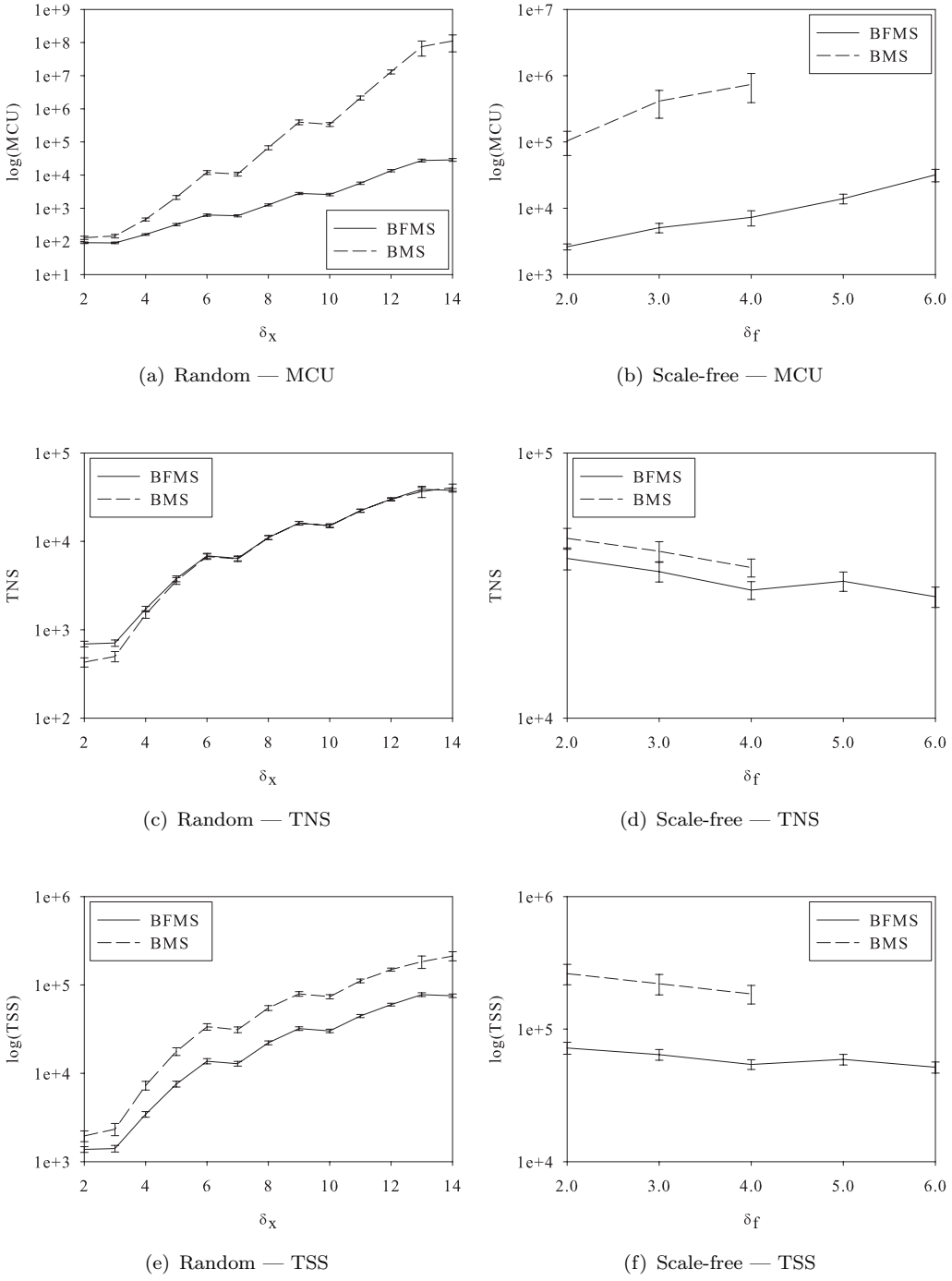
FIGURE 4.18: States pruned by ODP, with varying function shape, over increasing graph size.

with the scale of the graph, in order to show that BnB FMS meets our requirement of scalability. In more detail, to do this, we again generated 200 random graphs, where $|\mathcal{T}| \in \{50, 100, \cdots, 500\}$ and $|\mathcal{A}| = \frac{|\mathcal{T}|}{2}$, so that agents didn't all end up being connected to the same few tasks. We then generated links amongst agents and tasks to create random graphs, trees, and scale-free graphs, for average node degree $\delta_x \in \{3, 4, 5\}$. We ran ODP over all graphs, using random, superadditive and submodular function shapes, in order to evaluate how different utility function structures impacted the number of states pruned. Then, we measured the percentage of states that were pruned. Our results are plotted with 95% confidence intervals in Figure 4.18.

Figure 4.18 shows that, for random, superadditive, and submodular functions, the proportion of states pruned with ODP is scale invariant: it is effected only by the structure of the function (around 15% more states are pruned with a random function than with a superadditive, and around 40% of total states are pruned with a submodular function), not the size of the environment, as shown by the percentage of states removed being constant. This is because the number of states that can be pruned is controlled entirely by the shape of the functions in the environment — if the shape of the functions stays the same but the *number* of functions increases, this will have no effect on the percentage of states pruned, as is shown by these results. Thus, an interesting future direction for this work would be to investigate different functions to see what characteristics these functions show that always leads to the same proportion of states being removed at each agent.

4.5.2.2 Comparing BFMS and BMS

Now, Figure 4.19(a), (b), (e) and (f) show the marked reduction in computation and message size gained by the use of BFMS on both random and scale-free graphs: up to a maximum of 99% over BMS. This reduction in computation and communication is mostly due to our use of iGHS to reduce the complexity of the functions and variables by reducing the size of their domains. However, Figure 4.19(c) and (d) show that there



(a) Random — MCU

(b) Scale-free — MCU

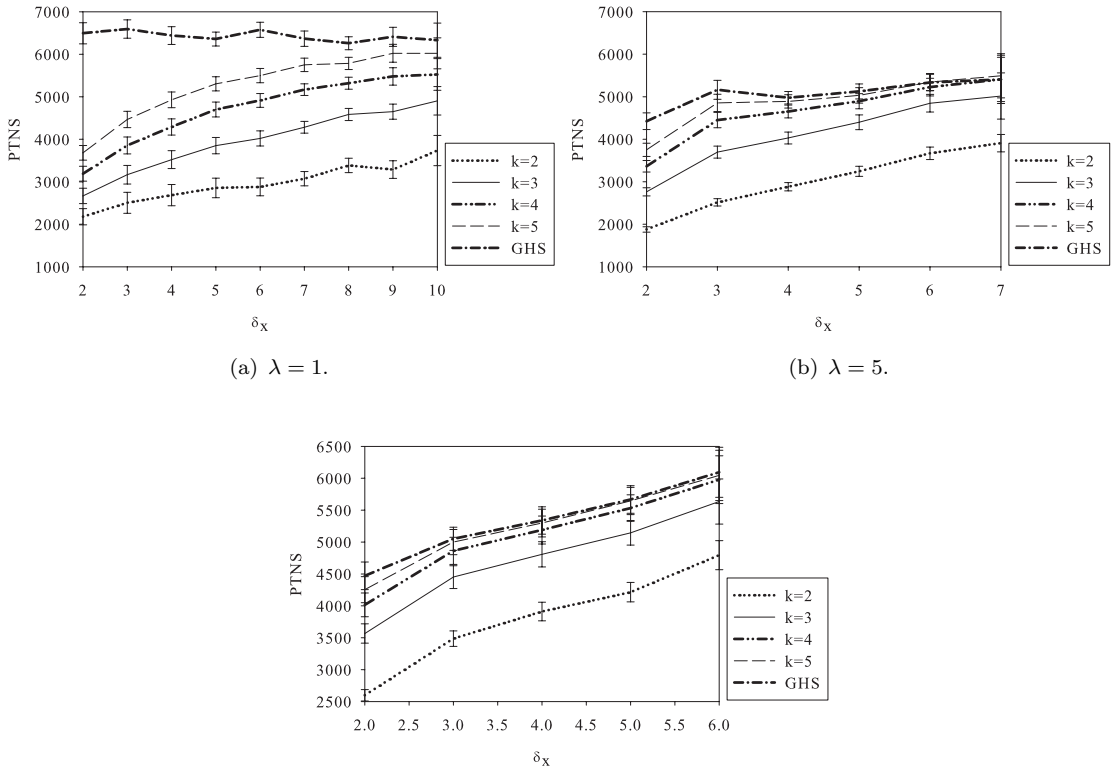
(c) Random — TNS

(d) Scale-free — TNS

(e) Random — TSS

(f) Scale-free — TSS

FIGURE 4.19: Comparing the robustness of BFMS and BMS on random and scale-free graphs, with $\lambda = 1$. Note that the BMS simulations did not complete beyond $\delta_f = 4$ on scale-free graphs.



(a) $\lambda = 1$.

(b) $\lambda = 5$.

(c) $\lambda = 10$.

FIGURE 4.20: PTNS achieved by BFMS with differing values of k and δ_x, compared to repeated application of GHS.

is not a significant reduction in the total number of messages sent by BFMS over the number sent by BMS when not including preprocessing messages. This is expected since BMS and BFMS are running a GDL algorithm over a tree, so the number of messages sent is likely to be similar. Note that we ran these experiments over differing λ and found that increasing the rate of change in the environment caused BFMS and iGHS to become equivalent to running BMS and GHS repeatedly, since a high rate of change leads to large portions of the solutions and graphs being recomputed.

Given this, we next show the impact that the value of k has on the number of preprocessing messages sent by BFMS. In doing this, and with the solution quality results which we will discuss in Section 4.5.3, we will explicitly show how the introduction of k in iGHS has produced a tradeoff between communication needed and solution quality achieved by BFMS. In more detail, we expect that lower values of k will have lower PTNS, at the expense of lower AR. Conversely, higher values of k, and re-running GHS as BMS does will result in higher AR, at the expense of higher PTNS.

To evaluate the impact of k on PTNS, we ran GHS and iGHS with $k \in \{1, 2, 3, 4, 5\}$ on the 200 random and scale-free graphs generated earlier, and aggregated the average total size of all iGHS messages sent by all nodes, measured in bytes (which we denote

PTNS).

We present our results in Figure 4.20: note that we only plot results for random graphs, since the scale-free results were very similar. The results in the Figure were as expected — lower values of k incur lower PTNS than higher values. In addition to this, higher values of λ lead to narrower differences in PTNS, since more changes occurring at once will lead to larger portions of the graph being recomputed — this is particularly noticeable in Figure 4.20(c), where $k = 5$ and GHS are almost indistinguishable. We show how the approximation quality achieved by iGHS is impacted by k later on, in Section 4.5.3.

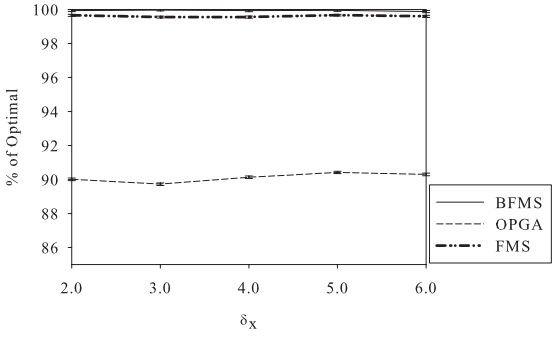
In the next section, we demonstrate how the solution quality achieved by our algorithms is impacted by graph density and scale.

4.5.3 Experiment 2: Solution Quality

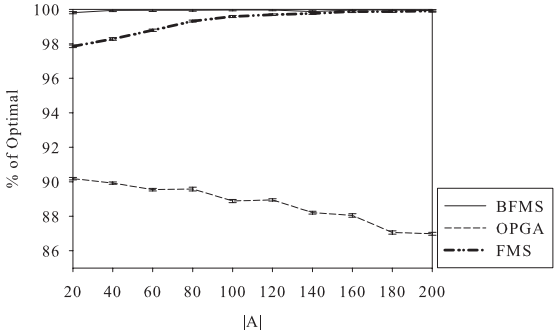
In this experiment, we aim to assess how the solution quality obtained by BFMS varies with increasing average node degree, number of agents, and number of tasks, as compared to FMS, an optimal algorithm (DPOP, see Section 2.2) and a greedy stochastic algorithm (OPGA, see Section 2.1.1). Thus, to do this, we varied graph density (i.e., the number of tasks that each agent can do) and scale (in terms of agents and tasks) in order to test how our algorithms meet our requirements of scalability (requirement 1), good solution quality (requirement 5), and boundedness (requirement 6).

We compare BFMS (with $k = 3$) against FMS and two other algorithms; namely, OPGA (see Section 2.1.1) which is the only other decentralised algorithm designed for a similar problem to ours, and DPOP (see Section 2.2) which, while earlier deemed unsuitable for our problem, will provide an optimal benchmark, thus allowing us to plot the quality of the solutions generated by BFMS, FMS and OPGA as a percentage of the optimal. All four algorithms were evaluated on the 200 instances (running $|\mathcal{T}| + |\mathcal{A}|$ iterations of BFMS on each), and the mean total utility obtained by each strategy as a percentage of the utility achieved by DPOP is shown in Figure 4.21, with 95% confidence intervals plotted as error bars.

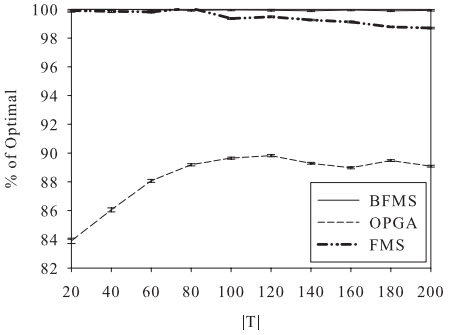
As can be seen from Figure 4.21(a), (b) and (c), FMS and BFMS outperform OPGA by up to 16%, over all δ_x, $|\mathcal{A}|$ and $|\mathcal{T}|$ tested. This is because OPGA only seeks to find local maxima, whereas BFMS finds the optimal of all considered solutions (and FMS approximates the optimal of all considered solutions on cyclic graphs). In addition to this, OPGA inherits a problem from the algorithm it is based upon: the distributed stochastic algorithm (DSA). More specifically, DSA and OPGA have the problem of agents sometimes thrashing between assignments. In more detail, we found that BFMS and FMS can achieve within 1% of the optimal utility found by DPOP given the settings of our experiments, compared to OPGA which is up to 16% worse. It is interesting to



(a) Varying node degree.



(b) Varying the number of agents.



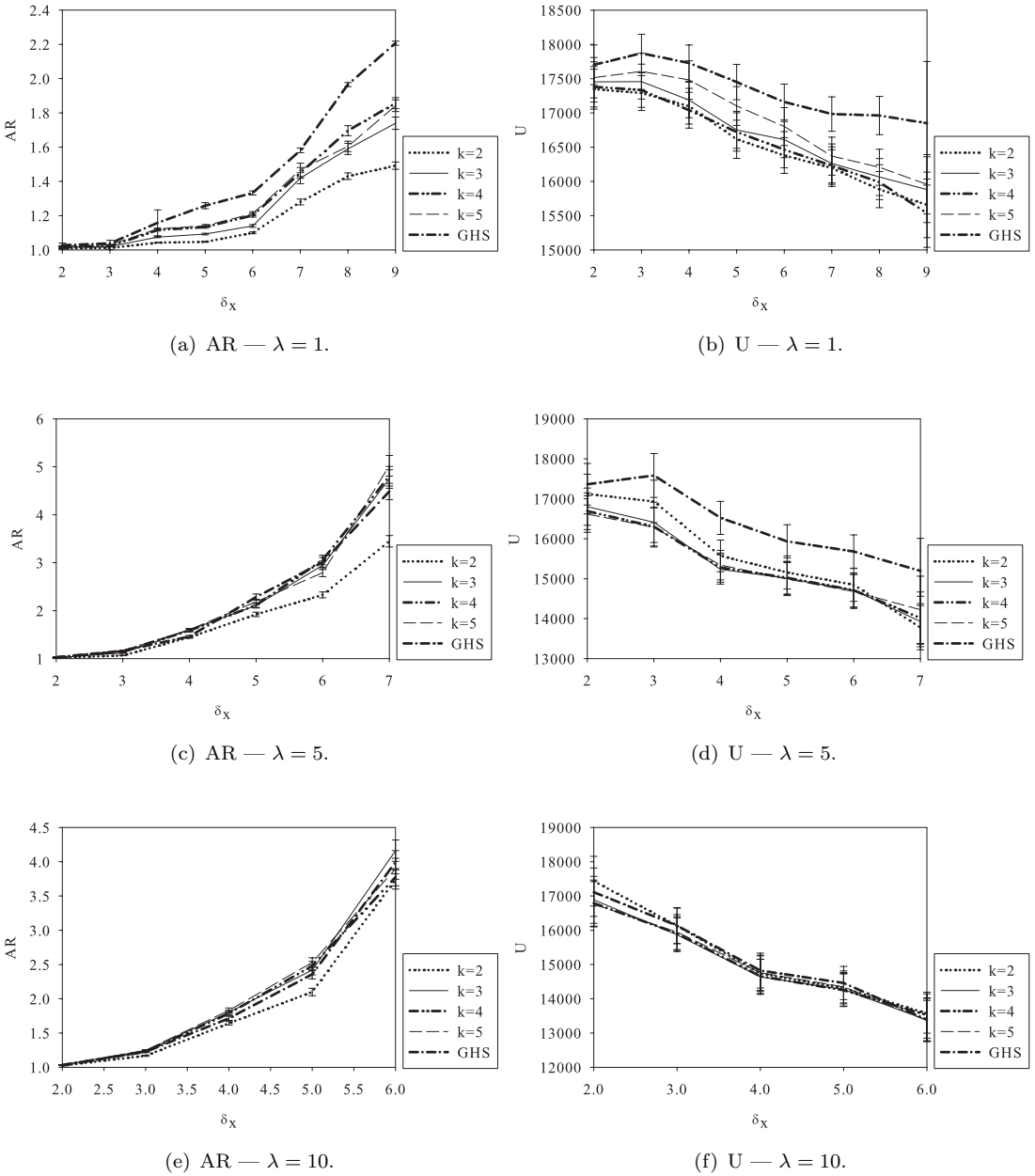
(c) Varying number of tasks.

FIGURE 4.21: Solution quality over varying node degree, and numbers of agents and tasks in random graphs.

note that OPGA and BFMS have opposite trends with increasing numbers of agents — this is because, whilst the problem becomes less difficult to solve with more agents (more agents to perform fewer tasks), the stochasticity of OPGA comes from the agents, so the more agents there are, the less deterministic the behaviour of the agents will be.

By comparing FMS to BFMS in these experiments, we see that the solution quality obtained by both algorithms is scale invariant in δ_x (and also in $|\mathcal{T}|$ and $|\mathcal{A}|$ for BFMS, which shows that the spanning tree found by iGHS contains good quality solutions). However, the solution quality achieved by FMS is negatively impacted at lower numbers of agents (because the impact of one variable having a suboptimal value is higher when there are few other variables), and higher numbers of tasks (where there is more potential for a variable to be set to a suboptimal task). In addition to this, we found that FMS consistently finds solutions within 2% of the quality achieved by BFMS on cyclic graphs. Thus, we have shown that even though FMS doesn't have provable solution quality or bounds on cyclic graphs, it still achieves good quality solutions.

In the previous section, we discussed the tradeoff introduced to BFMS by the parameter k — specifically, that lower values of k result in less communication but are likely to provide lower quality bounds (i.e., higher approximation ratios), and that higher values of k result in more communication but better quality bounds (i.e., lower approximation



(a) AR — $\lambda = 1$.

(b) U — $\lambda = 1$.

(c) AR — $\lambda = 5$.

(d) U — $\lambda = 5$.

(e) AR — $\lambda = 10$.

(f) U — $\lambda = 10$.

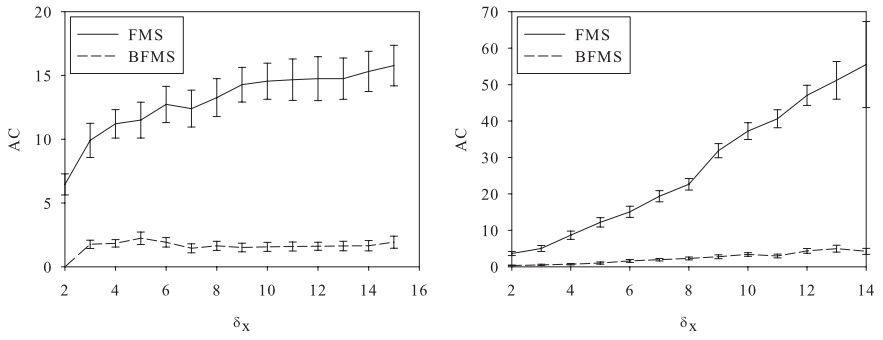
FIGURE 4.22: Average approximation ratio and utility achieved by BFMS with differing values of k and δ_x, compared to repeated application of GHS.

ratios). In Figure 4.22, we show that the results we obtained (using the settings we gave for the robustness experiment in Section 4.5.2) are not quite as expected, since they show that k doesn't have a very significant impact on AR (approximation ratio) — for all three values of λ, $k = 2$ is clearly tighter, since the graph doesn't change significantly enough to change the approximation ratio much, but $3 \leq k \leq 5$ and GHS are not significantly different to each other. This is because, where we have multiple changes each timestep in these experiments, the larger values of k will lead to similar graphs to re-run iGHS over, since larger subgraphs are likely to merge before GHS is run, which will produce similar STs, and so, similar results. Whilst this doesn't completely meet our expectations, it does show that the selection of the value of k is domain-specific, since in this particular domain the best value of k in terms of both AR and PTNS is $k = 3$.

A key observation here is that when $\lambda = 1$ (Figure 4.22 (a)), the approximation ratio found using GHS degrades more quickly with δ_x than that found using iGHS. Thus, the solution quality obtained by BMS will degrade in the same way. This is because the approximation ratio found by BMS is only proven to be tight on where functions are binary (i.e., where node connectivity is low), and is unproven beyond this. In contrast, the approximation ratio found by iGHS inherently will decrease more slowly, because only a section of the ST is recomputed at a time. However, see in Figure 4.22(b) that, despite this degradation in AR, running BFMS with GHS achieves markedly higher utility than using iGHS. This, therefore, means that using GHS results in a looser bound than using iGHS, but produces better quality results, because the quality of the solutions contained in the ST found by GHS are higher.

4.5.4 Experiment 3: Competitive Analysis

Here, we perform competitive analysis, as discussed earlier in Section 2.2.5.3. Here we consider competitiveness in terms of variable re-assignment: in most task allocation problems there is some cost of re-assigning agents to tasks, since the agent will likely have to stop whatever task they are doing, and physically move in the environment to get to the new one. Other measures could be used for this, for example communication and computation; however, we feel that variable reassignment is the more interesting and relevant for our problem given the physical side of real-world task allocation. An example of the worst-case scenario in terms of variable reassignment is where the environment undergoes so many changes that agents continually change their assignments, and spend their entire time moving between these assignments and never progressing in any tasks. Thus, given an environment which undergoes a number of changes in a finite time period (10 timesteps), the adversary (to use competitive analysis terminology) would produce a sequence of changes to the environment which would cause the algorithms to change variable assignments. Now, the optimal response to this would, with full information



(a) δ_x (b) δ_f

FIGURE 4.23: Results gained comparing the competitiveness of FMS and BFMS in terms of variable reassignments, where $\lambda = 1$.

about upcoming changes, perform only the assignments present at timestep 10. Hence, to assess the competitiveness of our algorithms, here, we record the number of variable reassignments performed by our algorithms over all 10 timesteps, and compute their competitive ratio on the basis that the optimal solution would perform no reassignments at all.

To do this, we used the same experimental settings as in our robustness experiments (see Section 4.5.1). We ran FMS and BFMS with $k = 3$ at each timestep, and recorded the total number of variable reassignments over all 10 timesteps. Given all this, we hypothesise that, in general, BFMS will have a better competitive ratio than FMS, due to iGHS 'dampening' the effect of changes in the environment by only recomputing sections of the ST.

We present our results in Figure 4.23, which demonstrates the relationship between the average degree of each type of node and the number of variable reassignments performed by FMS and BFMS. Note that we only reproduce the results for $\lambda = 1$ in the figure, since the other two values of λ produced very similar results. We give the minimum and maximum improvements in AC that BFMS gives over FMS in Table 4.3.

	$\lambda = 1$	$\lambda = 5$	$\lambda = 10$
δ_x	80–100%	89–100%	93–100%
δ_f	89–92%	90–97%	91–98%

TABLE 4.3: Minimum and maximum improvements given by BFMS over FMS, in terms of AC.

It is clear from these results that, as we expected, BFMS is far more competitive (80–100%) than FMS when speaking in terms of variable reassignments, even on the most difficult scenarios with many changes happening each timestep (i.e., when $\lambda = 10$)). In addition to this, Figure 4.23(a) and (b) show that the number of variable assignments performed by BFMS only increase very slightly with increasing δ_x and δ_f, due to the localising impact that iGHS has on changes. In contrast to this, FMS increases partic-

ularly sharply with δ_f — this is because when functions are more densely connected, the addition or removal of a single function impacts more variables, which recompute and trigger the re-computation of the variables and functions around them, thus making variable reassignment more likely.

4.6 Summary

We can summarise our results as follows. Specifically, in Section 4.5.2 we showed that FMS cuts the total message size sent by the max-sum algorithm by up to 99% (in the best case), and the total computation needed by up to 99% (in the best case), when changes are made to the underlying environment. Our algorithm also was also shown to achieve solutions with 99% of the utility of the optimal solution (see Section 4.5.3). However, whilst our algorithm gives benefits in communication and computation, it will not operate without the use of storage within the variable nodes in the factor graph. Nevertheless, this storage is linear in the number of edges and agents (see Equation 4.5), and so is kept to a minimum.

Next, we have shown that BnB FMS finds solutions of the same good quality as FMS (see Section 4.5.3), but requiring up to 99% less computation and up to 58% fewer messages than FMS in dynamic environments (see Section 4.5.2). However, this comes at the expense of a preprocessing step, and so BnB FMS should be used in large-scale environments, and FMS in smaller scale environments.

Finally, in BFMS, we have an algorithm which cuts the communication and computation done by BMS by up to 99% (Section 4.5.2), whilst still obtaining within 1% of the optimal utility (Section 4.5.3), when changes occur in the environment. This comes at the cost of storage at each agent, but enhances the FMS algorithm to obtain bounded approximate solutions over factor graphs containing cycles. Additionally, BFMS introduces a tradeoff between the value of k used in iGHS and the approximation quality — lower values of k sacrifice some of the solution quality in order to incur lower overheads. Given all this, BFMS is the algorithm to use if quality guarantees are needed — however, since the guarantees come at the expense of preprocessing, if quality guarantees are not required, then BnB FMS should be used in large-scale environments, and FMS should be used in smaller-scale environments.

Note that our approach to extend max-sum can be adapted to other types of problems where domain-specific properties can be exploited to reduce the state space. In more detail, the extensions we present in FMS can be generally applied to any domain where functions show a similar dependency structure from the variable. That is, the function has a significant change only for particular values of the domain (e.g., when the variable is allocated to the task that the function represents in our case). This clearly has a significant impact on the computation that factors go through when performing the

maximisation step, as we reduce the domain size of the variable. In particular, for our domain, a factor that depends on n variables that have a domain composed of d values each, will need to perform d^n computations, while with our extension this reduces to 2^n.

In what follows, we evaluate the contributions of the algorithms in this section with respect to the requirements posed in Section 1.3. Specifically:

1. **Scalability** — Similar to max-sum, FMS is an asynchronous message passing algorithm and only relies on peer to peer interactions, and is therefore scalable. However, a linear amount of storage is needed at each variable node, which will impact scalability slightly. BnB FMS specifically caters for this requirement, as the domain pruning and branch-and-bound elements of the algorithm work to reduce the computation that needs to be done at each agent. This, then, impacts the overall scalability of the algorithm as it reduces the computation that needs to be done by the system as a whole, and, as we showed in our evaluation, scales to hundreds of agents. Finally, as BFMS is built on FMS, it too is scalable because it is an asynchronous message passing algorithm and only relies on peer to peer interactions. However, the use of iGHS with BFMS incurs slightly more storage at each node than FMS. This only increases linearly in the number of nodes and connections in the graph.

2. **Robustness** — As required, in FMS and BnB FMS, decision making is spread throughout the system. The removal of one or more agents will not significantly impact how other agents make their decisions, as we show in Section 4.2.3. In BFMS, any changes to the environment will only effect the decision making of agents within k distance of that change.

3. **Efficiency** — By reducing both the amount and size of messages sent after disruptions, we have shown that FMS, BnB FMS, and BFMS significantly reduce the size of messages sent during their operation, by up to 99% less than max-sum and BMS. Thus, FMS, BnB FMS, and BFMS specifically address our requirement of efficiency in communication and computation.

4. **Adaptiveness** — FMS has been shown to specifically cater for a dynamic environment, and so recovers more efficiently from changes in the environment (and subsequently, changes in the factor graph) than max-sum. More specifically, our results show that when a change occurs in the factor graph, FMS uses up to 99% less computation on average than max-sum and reduces the message size sent by up to 99%. BnB FMS and BFMS also incur lower communication and computation overheads than BnB MS and BMS after changes in the environment. More specifically, BnB FMS uses up to 58% less communication on average than FMS, whilst still obtaining the same utility as FMS, and BFMS uses up to 99% less computation on average than BMS, whilst still obtaining within 1% of the optimal utility found by DPOP.

5. **Quality** — We have shown that the solutions found by our algorithms are of generally of good quality, and so, we consider this requirement satisfied by FMS, BnB FMS and BFMS.

6. **Boundedness** — We can only give quality guarantees for FMS on acyclic graphs (trees). In more detail, we can guarantee that FMS will converge to an optimal solution on tree-structured graphs, but limited theoretical results exist as to the convergence of GDL algorithms on cyclic graphs. BFMS provides particular advances over FMS in this area. Specifically, BFMS can be applied to any arbitrary graph and provide quality guarantees on the solutions provided.

	FMS	BnB FMS	BFMS
Scalability	✓	✓✓	✓
Robustness	✓	✓	✓
Efficiency	✓✓	✓✓	✓
Adaptiveness	✓✓	✓	✓
Quality	✓	✓	✓
Boundedness	×	×	✓✓

TABLE 4.4: Outline of how our contributions in this chapter relate to the requirements defined in Section 1.3. Here, we denote a requirement not satisfied by ×, and a requirement (strongly) satisfied by ✓(✓✓).

In Table 4.4, we provide an summary of how FMS, BnB FMS and BFMS meet our requirements. In more detail, FMS meets Requirements 1, 2, 3, 4, and 5 of the above: FMS is scalable, robust to agent failure, more efficient in terms of computation and communication than max-sum, is adaptive to changes in the environment, and finds good quality solutions. BnB FMS specifically addresses Requirements 1 and 3 by further reducing the communicational and computational costs of FMS, thus making it better suited to larger domains. Finally, BFMS specifically addresses Requirement 6, by providing quality guarantees on graphs where FMS is not proven to find the optimal solution. Thus, in combining the three algorithms, we meet all six of the requirements outlined in Section 1.3. However, despite this, we still need to solve the computation distribution problem described in Section 3.5 — we present our algorithms for doing this in the next chapter.

Chapter 5

Solving the Decentralised Computation Distribution Problem

In this chapter, we describe our algorithm for efficiently distributing factor computation in FMS and its variants, thus allowing us to meet our requirement of robustness. One key element of the algorithm is the use of min-max, which is another member of the GDL family mentioned in Chapter 2. In particular, we show that min-max has some unique properties, which stem from the idempotency of min and max, that differentiate it from other GDL algorithms such as max-sum or sum-product.

Building on this result, here we present ST-DTDA, which consists of four steps: (1) a preprocessing step which finds an initial solution to use as a starting point, (2) using heuristics to build an ST to improve the tractability of min-max, (3) running min-max on the ST to find an approximate solution, and (4) computing the approximation ratio of our solution. Thus, in these four steps, ST-DTDA uses localised message passing through min-max to find good quality, per-instance bounded approximate solutions in a distributed, efficient manner. This will then allow the computation of FMS to be distributed in a more efficient way than, for example, randomly assigning the computation, which could end up with a slower agent being given all of the computation, thus increasing the overall algorithm runtime.

We begin the chapter by reiterating some basic definitions from Chapter 3.2, expanding on some of them to facilitate description of the algorithm. We then provide a description of the min-max algorithm, and show how we can apply some simplifications to it that cannot be applied to other GDL algorithms. Next, we introduce decentralised task distribution algorithm (DTDA), which is an initial step toward decentralised distribution of computation, and provide a worked example of its operation. However, results from DTDA are poor in terms of computational complexity, and so we introduce ST-DTDA

to combat that, and, again, provide a worked example of its execution. Finally, we provide an empirical evaluation of ST-DTDA.

5.1 Basic Definitions

Here we recall some basic definitions from Section 3.2. Our domain consists of a set of agents $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$, which must compute a set of factor graph function nodes, $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$, each of which represent a task in the task allocation domain. The estimated amount of time it takes agent a_i to compute function f_j is denoted $\chi_i(f_j)$.[1] The challenge is then to assign the computation of each function to an agent such that the latest agent finish time (known as the *makespan*) is minimised. Here, we assume that the time it takes agent a_i to perform task t_j is completely unrelated to the time it takes another agent a_k to perform the task — thus, the agents' computation times are *unrelated*. Now, when thinking in terms of computing function nodes for FMS, the agent computation times are, in reality, likely to be related to each other somehow, however, in this work we generalise to unrelated agents, which is a more difficult problem. Thus, since related agents is a specialised version of unrelated agents, if we solve the generalised problem with unrelated agents, then by extension we have also solved the problem on related agents. Continuing on from this, we found that this unrelated agents problem is paralleled by a scheduling problem called scheduling on unrelated parallel machines, or $R||C_{\max}$. Thus, if we solve $R||C_{\max}$ in a decentralised manner, then we solve the more specialised computation distribution problem at the same time, whilst providing a meaningful contribution to operations research.

In order to solve this problem in a distributed manner, we formulate the domain as a junction graph as described in Section 3.5.2, where function node f_j is represented by a variable $y_j \in \mathcal{Y}$, each agent a_i is represented by a clique $Y_i \subseteq \mathcal{Y}$, and each agent's clique contains the variables representing the functions that the agent can compute for (i.e., $Y_i = \{y_k | f_k \in F_i\}$). Two cliques Y_i and Y_k are joined by an edge in the junction graph if and only if their variables overlap, and, if so, the edge will be labelled with the intersection of their variables, denoted $Y_{ik} = Y_i \cap Y_k$. Each clique is associated with a potential function $\psi_i(\mathbf{Y}_i)$, which returns the amount of time it will take a_i to compute the functions it has been assigned in variable assignment $\mathbf{Y}_i$.

Thus, the objective function is to find the variable assignment $\mathbf{Y}^*$ such that:

$$\mathbf{Y}^* = \arg\min_{\mathbf{Y}} \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i) \tag{5.1}$$

Given all this, in the next section we describe the algorithm which we use to solve the computation distribution problem, and show how the idempotency of the min and max

[1] As discussed earlier, it is difficult to get an exact measure of this given some computation relies on having received messages from elsewhere in the graph.

operators allows us to simplify some standard GDL computations.

5.2 The Min-max Algorithm

Like max-sum, min-max is a member of the GDL family of algorithms (Aji and McEliece, 2000). We know from the literature (for example, Rogers et al. (2011)) that GDL algorithms are decentralised, efficient in communication and computation (particularly in sparse graphs — in our case, where each agent can only perform a subset of the functions), provide generally good solutions, and have the potential to be adapted to respond to changes in the environment (as shown in FMS — see Chapter 4). Therefore, it makes sense to apply one here. In addition, GDL algorithms are proven to converge to optimal solutions on acyclic junction graphs (junction trees). Thus, the algorithm is guaranteed to complete such that all participating agents end up with the same solution on acyclic graphs. In more detail, GDL algorithms are all based on a commutative semiring and min-max is based on the commutative semiring $\langle \mathbb{R}^+, \min, \max, \infty, 0 \rangle$ where min is the additive operator and max the multiplicative operator.

Given a junction graph (as formulated in Section 3.5.2), min-max consists of exchanging messages between agents and their neighbours in the junction graph. Let $Y_{ij} = Y_i \cap Y_j$ be the intersection of variables of two neighbours, a_i and a_j. In min-max, agent a_i exchanges messages with a neighbour $a_j \in \mathcal{N}(a_i)$ containing the values of a function $\mu_{i \to j} : Y_{ij} \to \mathbb{R}^+$.

Initially, all such functions are defined to be 0 (the semiring's multiplicative identity). Once messages have been received, the message is computed as follows:

$$\mu_{i \to j}(\mathbf{Y}_{ij}) = \min_{\mathbf{Y}_{i \setminus j}} \max \left[\psi_i(\mathbf{Y}_i), \max_{a_k \in \mathcal{N}(a_i) | k \neq j} \mu_{k \to i}(\mathbf{Y}_{ki}) \right] \tag{5.2}$$

where $Y_{i \setminus j} = Y_i \setminus Y_j$ and $\mathbf{Y}_i$ and $\mathbf{Y}_{ki}$ stand for the projection of $\mathbf{Y}_{ij}$;$\mathbf{Y}_{i \setminus j}$ over variables in Y_i and Y_{ki} respectively. In computing this message, agent a_i iterates through the combined state space of the variables in Y_i, excluding any variables shared with Y_j, in order to find the smallest makespan estimate, both from its own function and from the messages it has received. In order to compute the makespan estimate for each configuration of variables, the agent takes the maximum of its own function value for a configuration, and the biggest message value it has received from a neighbour (other than a_j) for that configuration.

Similarly, for each clique a_i, min-max computes an approximation of the marginal contribution of its variables, $\tilde{\zeta}_i \approx \zeta_i$, where $\tilde{\zeta}_i : \mathbf{Y}_i \to \mathbb{R}^+$, as:

$$\tilde{\zeta}_i(\mathbf{Y}_i) = \max \left[\psi_i(\mathbf{Y}_i), \max_{a_j \in \mathcal{N}(a_i)} \mu_{j \to i}(\mathbf{Y}_{ji}) \right] \tag{5.3}$$

In the particular case of a junction tree, this approximation of the marginal contribution is guaranteed to be exact, so the solution found by min-max is guaranteed to be optimal, which means $\tilde{\zeta}_i(\mathbf{Y}_i) = \zeta_i(\mathbf{Y}_i)$ for all $a_i \in \mathcal{A}$ and all $\mathbf{Y}$. This computation is almost the same as computing the makespan estimates in the message computation described above — to compute its marginal contribution for configuration $\mathbf{Y}_i$, agent a_i takes the maximum of its own function value for a configuration, and the biggest message value it has received from *any* neighbour for that configuration. Now, the idempotency of max, the multiplicative operator (Rossi et al., 2006), allows us to make a number of changes to the standard GDL formulation, which we explain next.

5.2.1 Simplifications to GDL Computations

Idempotency implies that, for all $r \in \mathbb{R}^+$, $\max(r, r) = r$. Hence, the idempotency of the multiplicative operator implies that repeatedly combining the same information will not produce new information, or lose any relevant old information. Moreover, when an operator is idempotent, it defines a partial ordering over the set $\mathbb{R}^+$. In our case, both operations are idempotent. For the min operator, the order is the natural order of real numbers: i.e., $r \leq s$ if and only if $\min(r, s) = r$. Conversely, for the max operator, the order is the inverse of the natural ordering of numbers: i.e., $r \geq s$ if and only if $\max(r, s) = r$. From these, we can deduce that, as min orders the elements in exact inverse to max, $\min(r, \max(r, s)) = r$, and therefore $\max(r, \min(r, s)) = r$.

Due to the idempotency of the min-max commutative semiring, the following equality holds for any Y_i, Y_i', $\mathbf{Y}_i$ where $Y_i' \subseteq Y_i$:

$$\max\left[\psi_i(\mathbf{Y}_i), \min_{\mathbf{Y}_i'} \psi_i(\mathbf{Y}_i')\right] = \psi_i(\mathbf{Y}_i) \tag{5.4}$$

Thus, the smallest value of ψ_i for any configuration of a subset of the variables in Y_i will always be smaller than or equal to the value of ψ_i for any given configuration of Y_i. This means that repeatedly combining messages from other agents will not change the outcome of min-max.

This idempotency property is a feature we exploit in our implementation of min-max, to improve efficiency. In more detail, the idempotency of the min-max semiring, a property not shared with other non-idempotent GDL semirings, allows us to *simplify* Equations 5.2 and 5.3 such that:

- in Equation 5.2, when an agent a_i sends a message to a neighbour a_j, it does not need to marginalise out any previously received messages from a_j. This is because the result will be the same as it would be when eventually a_j's messages are reflected in those received from other agents.

- in Equation 5.3, the agent's marginal contributions can be computed recursively by combining the messages it has sent from multiple iterations, which, again, reduces computation at the agent by removing the need to iterate over all neighbours' messages, as required by Equation 5.3. Thus, the computation is reduced from $O(n)$ to $O(1)$, as the agent only needs to compare its function value with the previous message it sent for that value. This is because repeatedly combining messages from previous iterations will not change the approximate marginal contribution at an agent.

In addition to these computational simplifications, the idempotency property provides two further properties that make the min-max algorithm more efficient than non-idempotent GDL algorithms that are based on functions: (1) it is guaranteed to converge even in cyclic graphs (as we prove with Theorem 5.1), and (2) it provides an online per-instance bound on the optimal solution value of the problem that it approximates (see Section 5.3.5 for more details). As mentioned earlier, guaranteed convergence is important in message passing algorithms because there is a guaranteed end to message passing, and every agent is guaranteed to come to the same solution (or set of solutions) at the end of the algorithm's execution.

Theorem 5.1. *The min-max algorithm is guaranteed to converge in a finite number of iterations.*

Proof. Bistarelli et al. (2000) prove the termination of idempotent commutative semirings (Theorem 8). Given the fact that min-max is an idempotent semiring, the min-max algorithm must terminate. □

Now that we have explained the useful properties of min-max, in the next section, we present DTDA, which is our first step towards finding a decentralised algorithm for $R||C_{\max}$. DTDA consists of an algorithmic instantiation of min-max, which, when combined with a value propagation phase, allows online per-instance bounds on solution quality to be obtained at each agent.

5.3 Using Min-max for Computation Distribution

Broadly speaking, DTDA consists of two key steps:

1. **Applying the min-max algorithm**: to propagate information across the agents to produce a set of solutions.

2. **Value propagation**: to ensure all agents choose the same assignment of shared variables, and to compute the per-instance bound on the quality of that assignment. Agents are arranged in a tree structure, and each agent selects a solution for its clique's variables that is consistent with all other agents' solutions.

We elaborate on these steps in the following subsections.

5.3.1 Phase 1: Applying Min-max

In the first step of the DTDA, we apply min-max over the junction graph described in Section 3.5.3, in order to find a set of solutions (allocations of functions to agents).[2]

We present the pseudocode for min-max in Algorithm 5.1. Thus, an agent begins by running the procedure `initialise` (lines 1–5). Each agent starts by initialising its stored outgoing messages to 0 (line 2), and its marginal contribution function, $\tilde{\zeta}_i$, to the agent's potential function, ψ_i, which encodes the agent's own cost function, computed as given in Equation 3.5 (line 3).

After initialisation, each agent exchanges a message, $\mu_{i\to j}$, with each of its neighbours, $a_j \in \mathcal{N}(a_i)$, in order to find out their marginal contribution for each configuration of the variables they share. This is done via the procedure `send_messages` (lines 13–18). The message $\mu_{i\to j}(Y_{ij})$ is sent over all combinations of the variables in Y_{ij} (i.e., the intersection of Y_i and Y_j). The content of the message from an agent a_i to a_j is, therefore, agent a_i's marginal contribution function, computed as in Equation 5.3:

$$\mu_{i\to j}(\mathbf{Y}_{ij}) = \min_{\mathbf{Y}_{i\setminus j}} \tilde{\zeta}_i(\mathbf{Y}_i) \tag{5.5}$$

When an agent a_i receives a message $\mu_{j\to i}$, it runs the procedure `received` (lines 6–12), in which the agent checks if the message it has received differs from the last message it received from that agent. This is important to ensure that the messages stop being sent when they stop changing, so the algorithm converges to a solution.[3] If the message does differ (line 7), then a_i updates its *stored* entry for the sending agent (line 8). Afterwards, the agent a_i updates its marginal contribution values (line 9) as follows:

$$\tilde{\zeta}_i(\mathbf{Y}_i) = \max\left\{\tilde{\zeta}_i(\mathbf{Y}_i), \mu_{j\to i}(\mathbf{Y}_{ji})\right\} \tag{5.6}$$

[2]Note that we specify that a set of solutions is produced, because it is possible for more than one solution to have the same value. This is because the solution value is taken to be the largest makespan at one agent — therefore, many allocations of functions and agents could give the same makespan. This is solved in step 2 (Section 5.3.2), where we use value propagation to choose one solution.

[3]We assume that the algorithm has converged after no messages have been received at an agent for a pre-determined length of time known as a *timeout*. The exact length of this timeout will differ depending on the size of the environment and the latency of the communication links amongst the agents, to ensure that the agents do not falsely assume the algorithm has converged when it has not.

Algorithm 5.1 `min-max()` at agent a_i.

1: **procedure** `initialise`
2: Initialise messages $\mu_{i\to j}(\mathbf{Y}_{ij}) = 0 \quad \forall j \in \mathcal{N}(i)$
3: $\tilde{\zeta}_i(\mathbf{Y}_i) = \psi_i(\mathbf{Y}_i) = \sum_{y_k \in \mathbf{Y}_i, y_k = i} \chi_i(f_k)$ `// Initialize marginal contribution`
4: Run procedure `send_messages` (line 13).
5: **end procedure**

6: **procedure** `received` $\mu_{j\to i}$
7: **if** $stored(j) \neq \mu_{j\to i}$ **then** `// Received different message`
8: $stored(j) = \mu_{j\to i}$ `// Update stored message`
9: Recompute $\tilde{\zeta}_i(\mathbf{Y}_i)$
10: Run procedure `send_messages` (line 13).
11: **end if**
12: **end procedure**

13: **procedure** `send_messages`
14: **for** $j \in \mathcal{N}(i)$ **do**
15: Recompute $\mu_{i\to j}(\mathbf{Y}_{ij})$
16: Send $\mu_{i\to j}(\mathbf{Y}_{ij})$ to a_j `// Send min-max message to neighbour`
17: **end for**
18: **end procedure**

This marginal contribution is not exact because GDL algorithms are only guaranteed to compute exact solutions on junction trees and not on general graphs (Aji and McEliece, 2000). Finally, agent a_i re-sends all of its messages (line 10).

These messages are passed amongst agents until their content no longer changes — at which point, each agent will ascertain the best states for its variables. Next, we describe our value propagation phase that agents run once messages have converged. This aims to ensure all agents individually choose the same solution for their shared variables from the set of solutions they have found, and compute the per-instance bound of the approximate solution.

5.3.2 Phase 2: Value Propagation

Once the messages amongst agents have converged,[4] and no further messages need to be sent, we introduce a two-part value propagation phase to ensure the agents all set the values of their shared variables to the same values, and are aware of the quality of their solution. This is required due to the likelihood of multiple solutions being present, and to facilitate the computation of the per-instance bound given in Section 5.3.5.

In the first part of this phase (see Algorithm 5.2, lines 1–10), we arrange the agents into a Depth-First Search (DFS) tree using a distributed DFS algorithm such as (Collin and Dolev, 1994). In particular, a DFS tree must always ensure that agents which are adjacent in the original graph are in the same branch. This ensures relative independence of nodes in different branches of the tree (i.e., two nodes in different branches of the tree will share only the variables of the node that joins their subtrees), allowing parallel propagation of values. For any given graph, our DFS tree is considered 'valid' if no ties (variable overlaps) between agents in different subtrees of the DFS tree exist. The outcome of this DFS is that each agent has set the values of its *parent* and *children* variables shown in Algorithm 5.2. Once this has occurred, the root node computes a configuration of its variables to propagate, $\mathbf{Y}_i^*$, which minimises the agent's marginal contribution:

$$\mathbf{Y}_i^* = \arg\min_{\mathbf{Y}_i} \tilde{\zeta}_i(\mathbf{Y}_i) \tag{5.7}$$

Hence, this equation provides an approximation of Equation 3.6 in general junction graphs, but is the exact solution in the particular case of junction trees. Having computed this, the root node sends $\mathbf{Y}_i^*$, along with $v_i = \psi_i(\mathbf{Y}_i^*)$ (the actual value of the current solution) and $\zeta_i(\mathbf{Y}_i^*)$ (the value of the current solution as computed by min-max) to the node's children. Each of these children adds their own variables onto $\mathbf{Y}_i^*$ (line 2), takes the maximum of v_i and ζ_i with what they have received (lines 3 and 4, respectively), and sends these new values onto their own children (line 5).

[4] As we said in Section 5.3.1, agents assume messages have converged after they haven't received any messages for a time dependent on the size of the environment.

Algorithm 5.2 `valueprop` at agent a_i

Require: *parent*, *children*

1: **procedure** `received`($\langle \mathbf{Y}_p^*, v_p, \tilde{\zeta}_p^* \rangle$) from *parent*
2: $\mathbf{Y}_i^* = \arg\min_{\mathbf{Y}_{i \setminus p}} \tilde{\zeta}_i(\mathbf{Y}_{i \setminus p}; \mathbf{Y}_p^*)$ `// compute best variable states`
3: $v_i = \max(v_p, \psi_i(\mathbf{Y}_i^*))$ `// compute current makespan`
4: $\tilde{\zeta}_i^* = \max(\tilde{\zeta}_p^*, \tilde{\zeta}_i(\mathbf{Y}_i^*))$ `// compute current marginal contribution`
5: Send $\langle \mathbf{Y}_i^*, v_i, \tilde{\zeta}_i^* \rangle$ to all $a_j \in$ *children*
6: **if** *children* $= \emptyset$ **then** `// is a leaf node`
7: Send $\langle v_i, \tilde{\zeta}_i^* \rangle$ to *parent*
8: **end if**
9:
10: **procedure** `received`($\langle v_p, \tilde{\zeta}_p^* \rangle$) from *child*
11: **if** Received messages from all *child* $\in$ *children* **then**
12: $v_i = \max(v_p, \psi_i(\mathbf{Y}_i^*))$ `// update makespan`
13: $\tilde{\zeta}_i^* = \max(\tilde{\zeta}_p^*, \tilde{\zeta}_i(\mathbf{Y}_i^*))$ `// update marginal contribution`
14: $\rho = \tilde{\zeta}_i^* / v_i$ `// compute approximation ratio`
15: Send $\langle v_i, \tilde{\zeta}_i^* \rangle$ to *parent*.
16: **end if**

Once this first phase is complete (i.e., the messages have reached the leaf nodes), the leaf nodes pass their marginal contribution and makespan values (the actual value of the solution) back up the tree (lines 6 and 7), to ensure every agent can compute the quality of the solution. Then, when an agent has received such a message from each of its children (line 11), it will update its v and $\zeta_i(\mathbf{Y}_i^*)$ values (lines 12 and 13, respectively), calculate its approximation ratio ρ (line 14) — which is the agent's per-instance bound (this will be clarified in section 5.3.5), and send the new v and $\zeta_i(\mathbf{Y}_i^*)$ values to its parent (line 15). Once these messages have reached the root of the tree, the DTDA is complete, all variables have values consistent across all agents, and all agents are aware of the quality of their solution.

5.3.3 Managing Disruptions

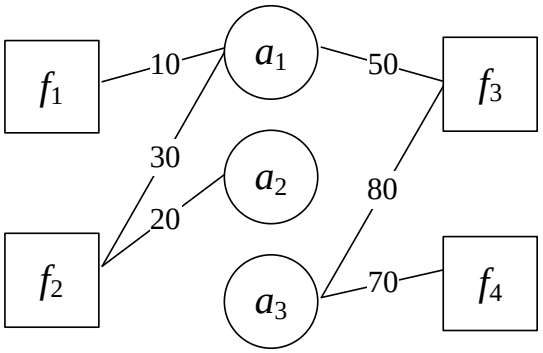
Earlier, we stressed that our algorithms must be applicable in *dynamic* environments which change over time. To respond to a change in the environment, we could simply re-run DTDA from scratch and compute a new distribution of functions to agents; however, as with FMS, we can store certain values at agents to ensure that repeated communication after a change in the environment is kept to a minimum. In more detail, at agent a_i, we store the last message received from each neighbouring agent, and the last state chosen for each of the variables in a_i's clique. Then, if an agent receives a message from its neighbour that hasn't changed, no further messages need to be sent. In the next section, we provide a worked example of the execution of DTDA, before going on to prove key properties of the algorithm.

5.3.4 Worked Example

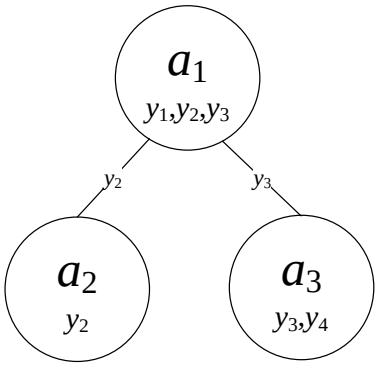
We present a sample computation distribution problem in Figure 5.1 (a), which is a tree-structured version of the graph given in Section 3.5. Our junction graph formulation of this problem is given in Figure 5.1 (b) — since this is a junction tree, we know that DTDA will find the optimal solution here.

To begin phase 1 of the algorithm, a_1 sends μ messages (as described in Equation 5.5) to a_2 and a_3 regarding their common variables (see Figure 5.1 (c)). a_2 and a_3 receive the messages, update their marginal contribution values as per Equation 5.6, and respond with their own μ messages to a_1 (see Figure 5.1 (d)), which updates its own marginal contribution value, and responds (see Figure 5.1 (e)).

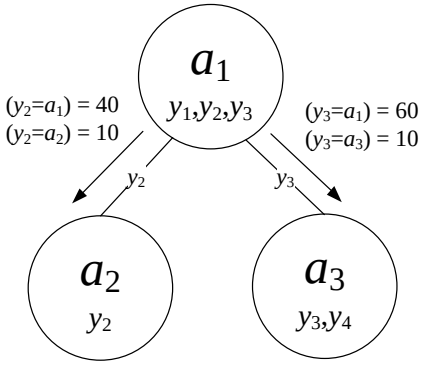
At this point, since the message received at a_3 from a_1 is no different to the message a_3 previously sent to a_1, a_3 can set its variable values to those that minimise $\zeta_3(\mathbf{Y}_3)$: specifically, $y_3 = a_1$ and $y_4 = a_3$, as shown in Figure 5.1 (f). In contrast, a_2 received a different message to the one it sent, so it recomputes and resends its message to a_1 (see



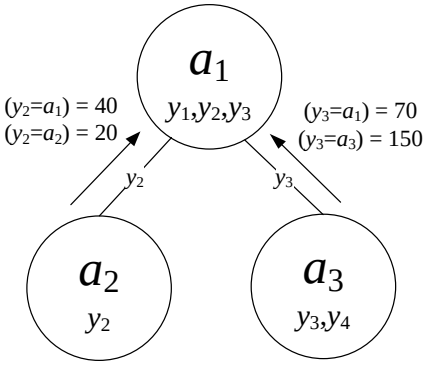
(a) Sample problem.



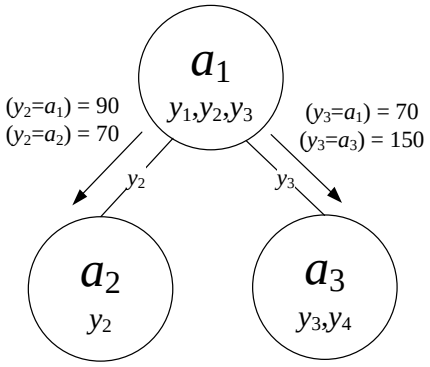
(b) Junction graph of problem.



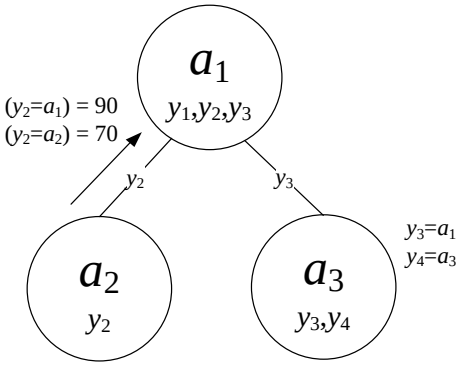
(c) a_1 sends messages.



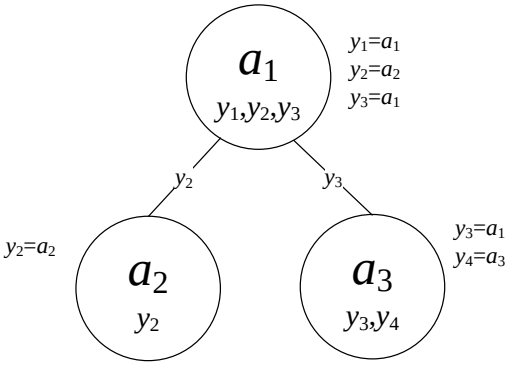
(d) a_2 and a_3 reply to messages.



(e) a_1 replies to messages.



(f) a_2 replies to messages, a_3 computes its variable values.



(g) a_1 and a_2 compute their variable values.

FIGURE 5.1: Worked example of the operation of DTDA on a tree-structured problem.

Figure 5.1(f)). This message is the same as the one sent by a_1 to a_2, so following this, a_1 and a_2 compute the best state for their variables, as marked in Figure 5.1 (g).

The second and final step of DTDA is the value propagation step, in which the agents arrange themselves into a DFS tree (which they already are in), and a_1 sends its $\mathbf{Y}_1^*$ and v_1 values to a_2 and a_3. a_3 will add y_4 into the solution (a_2 does not need to add any variables to the solution since a_1 will already have set the value of y_2), and a_2 and a_3 will then send their marginal contribution and makespan values back to a_1, after which all agents can compute the approximation ratio $\rho = \tilde{\zeta}_i^*/v_i$, which in this case will be $\frac{70}{70} = 1$.

5.3.5 Properties of DTDA

To prove the correctness of DTDA, we must prove that the bound we find holds. The key idempotency properties of min-max identified in Section 5.2 allow each agent to compute a per-instance bound on the quality of the solution gained, with very little computational overhead. In more detail, here, we prove that the maximum value of the solution that minimises the approximate marginal contributions of the agents in min-max is a lower bound on the cost of the optimal solution of the $R||C_{\max}$ problem, as formulated in Equation 3.6. This lower bound can then be used by agents to assess the quality of the solution found by min-max.

Before proving this claim, we define the relation of *equivalence* among two functions.

Definition 5.2 (equivalence)**.** Given two functions $\alpha(Y_\alpha)$ and $\beta(Y_\beta)$ we say they are equivalent if they: (1) are defined over the same set of variables $Y_\alpha = Y_\beta$; and (2) return the same values for all possible configurations, $\alpha(\mathbf{Y}) = \beta(\mathbf{Y})$. We denote such a relation of equivalence as $\alpha \equiv \beta$.

Next, we prove two lemmas, that help to prove Theorem 5.5. First, in Lemma 5.3, we state that, at any iteration τ of the min-max algorithm, the function that results from the combination of the agents' marginal contributions, namely $\tilde{Z}^\tau(\mathbf{Y}) = \max_{a_i \in \mathcal{A}} \tilde{\zeta}_i^\tau(\mathbf{Y}_i)$, is *equivalent* to the objective function in $R||C_{\max}$, given in Equation 3.6, which we denote here as minimising $\Psi(\mathbf{Y}) = \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i)$. Thus, under Lemma 5.3, $\tilde{Z} \equiv \Psi$. Second, in Lemma 5.4, we state that $\tilde{\zeta}$, defined as the maximum of the individual agents' marginal solutions, is a lower bound on the value of the optimal value of function $\tilde{Z}$.

We provide formal proofs for these two lemmas below.

Lemma 5.3. *At any iteration τ of the min-max algorithm, $\tilde{Z}^\tau(\mathbf{Y}) \equiv \Psi(\mathbf{Y})$.*

Proof. We prove this by induction on τ.

For $\tau = 0$ the case is trivial, $\tilde{Z}^0(\mathbf{Y}) = \max_{a_i \in \mathcal{A}} \tilde{\zeta}_i^0(\mathbf{Y}_i) = \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i) = \Psi(\mathbf{Y})$.

Then we prove $\tau = n+1$: that is, that $Z^{n+1} \equiv Z^n$, assuming that $\tau = n$ holds. $\tilde{Z}^{n+1}(\mathbf{Y}) = \max_{a_i \in \mathcal{A}} \max(\tilde{\zeta}_i^n(Y), \max_{a_j \in N(a_i)} \min_{\mathbf{Y}_{j \setminus i}} \tilde{\zeta}_j^n(\mathbf{Y}_{j \setminus i}))$. Since the max operator is commutative and associative, $\tilde{Z}^{n+1}(\mathbf{Y})$ can also be written as $\max_{a_i \in A} \max(\tilde{\zeta}_i^n(\mathbf{Y}_i), \max_{a_j \in \mathcal{N}(a_i)} \min_{\mathbf{Y}_{i \setminus j}} \tilde{\zeta}_i^n(\mathbf{Y}_{i \setminus j}))$. Then, by exploiting the idempotency of the max operator (see Equation 5.4), $\tilde{Z}^{n+1}(\mathbf{Y})$ simplifies to $\max_{a_i \in \mathcal{A}} \tilde{\zeta}_i^n(\mathbf{Y}_i)$ and $\tilde{Z}^{n+1} \equiv \tilde{Z}^n \equiv \Psi$. □

Lemma 5.4. *Let $\tilde{\zeta}^*$ be the value of the assignment x^* that minimises $\tilde{Z}(\mathbf{Y})$. Then, $\tilde{\zeta} = \max_{a_i \in \mathcal{A}} \min_{\mathbf{Y}_i} \tilde{\zeta}_i(\mathbf{Y}_i)$ is a lower bound on $\tilde{\zeta}^*$, $\tilde{\zeta} \leq \tilde{\zeta}^*$.*

Proof. We prove this by contradiction. Assume that there is an assignment $\mathbf{Y}$ of $\mathcal{Y}$ such that $\tilde{Z}(\mathbf{Y}) \leq \max_{a_i \in \mathcal{A}} \min_{\mathbf{Y}_i} \tilde{\zeta}_i(\mathbf{Y}_i)$. This leads to a contradiction, because it implies that at least one function $\tilde{\zeta}_i$ evaluated at y is lower than its minimum, $\min_{\mathbf{Y}_i} \tilde{\zeta}_i(\mathbf{Y}_i)$. □

Finally, we combine these two lemmas to prove our main result, in Theorem 5.5.

Theorem 5.5. *$\tilde{\zeta}$ is a lower bound on the value of the optimal solution, namely $\tilde{\zeta} \leq \min_{\mathbf{Y}} \Psi(\mathbf{Y})$*

Proof. Since the optimal solution of two equivalent functions is the same, the result follows directly from Lemmas 5.3 and 5.4. □

Therefore, under Theorem 5.5, at each iteration of the min-max algorithm, the maximum of the agents' marginal contributions, $\tilde{\zeta}$, is a lower bound on the value of the optimal solution. Notice that, at each iteration, the agents' marginal contribution functions combine information from the messages using the max operator, so $\min_{\mathbf{Y}_i} \tilde{\zeta}_i^\tau(\mathbf{Y}_i) \leq \min_{\mathbf{Y}_i} \tilde{\zeta}_i^{\tau+1}(\mathbf{Y}_i)$. Therefore, this lower bound is guaranteed to monotonically increase over iterations of min-max, thus providing a better approximation of the value of the optimal solution at each iteration. As shown in Section 5.3.2, agents can, at the end of the min-max algorithm, use this lower bound value to compute an approximation ratio for the solution found when running the min-max algorithm.

Next, we show that DTDA is superstabilizing (see Section 2.2.5.1) subject to the following predicates: *legitimacy*: $M(\mathbf{Y}) = \min_{\mathbf{Y}} \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i)$ on acyclic graphs, where $M(\mathbf{Y})$ is the makespan of the solution produced by DTDA (i.e., the variables must be in an optimal state), and *passage*: a valid joint variable assignment should be maintained at all times, even when a new solution is being computed. We prove this property below, following the proof of superstabilization we gave for FMS in Section 4.2.5:

Proposition 5.6. *DTDA is superstabilizing on acyclic junction graphs, with respect to the legitimacy and passage predicates given above.*

Proof. First, DTDA is an extension to min-max, which is proven to converge to an optimal solution on tree structured graphs in a finite number of messages (Mackay, 2003). Second, when a change occurs in the graph, DTDA is run again, and therefore, provided the change did not introduce a cycle into the graph, DTDA is guaranteed to converge to the optimal again, satisfying the legitimacy predicate, thus reaching a legitimate state within a finite amount of time. Now, DTDA is superstabilizing because it maintains each variable's previous valid state at each variable node at all times, only updating it after recalculation, and so, the passage predicate always holds. □

Finally, DTDA is an anytime algorithm (see Section 2.2.5.2) because each agent can compute a state for each variable it controls at any time during execution. In addition to this, the joint variable state will improve with each message-passing iteration of min-max.

5.3.6 Summary

Now, whilst DTDA is the first decentralised algorithm for $R||C_{\max}$, it suffers from one key weakness: its complexity. In more detail, the $\mu_{i\to j}$ messages are exponential in size and complexity: each message contains one computed value for each possible combination of the variables in $Y_i \cap Y_j$, and for each of these computed values, involves a minimisation over all possible combinations of $Y_{i\setminus j}$. This means that the complexity of DTDA is $O(d^n)$, where $d = \max_{a_i \in \mathcal{A}} |T_i|$ and $n = |\mathcal{A}|$, and so, in practice, DTDA is intractable, and so is not practical for use in our dynamic task allocation environments. We fix this with the use of preprocessing in the next section.

5.4 Using Preprocessing to Improve the Tractability of DTDA

As shown by algorithms such as BMS (Section 2.3.5) and BFMS (Section 4.4.2), adding a preprocessing step to a computationally complex algorithm can reduce computational complexity whilst allowing the quality of solutions produced to be bounded. Thus, in order to improve the tractability of DTDA, in this section we propose ST-DTDA which is able to provide an approximate solution in linear time. ST-DTDA requires less computation at each agent to obtain an allocation of agents to functions. Furthermore, in experiments, we show how ST-DTDA is able to provide solutions that are close to the optimal. In more detail, ST-DTDA uses a ST approximation of the problem algorithm by ignoring some potential function-agent assignments, at the cost of finding an approximate solution as opposed to an optimal one.

ST-DTDA consists of four steps:

1. Step 1: Preprocessing — in this step, we use a simple iterative distributed greedy algorithm to find an initial solution, which we use as a starting point for our heuristic-based spanning tree approach in step 2.

2. Step 2: Building a spanning tree — we use a distributed ST algorithm to find a tree of the graphical representation of the environment given in Section 3.5. We propose three different heuristics to weight edges of the original problem.

3. Step 3: Running the min-max algorithm — we use min-max to optimally solve the resulting tree structured problem, providing a bounded approximation specific to the particular problem instance.

4. Step 4: Computing the approximation ratio — we use message passing to compute a worst-case bound on the distance between the quality of the optimal solution and the quality of the solution to the tree-structured problem. We then use this error bound and a lower bound on the optimal solution value to estimate the approximation ratio.

We elaborate further on these steps in the remainder of this section.

5.4.1 Step 1: Preprocessing

In the first step of ST-DTDA, we run a simple iterative decentralised greedy algorithm (which is loosely based on DSA) at each agent to provide an initial solution for the heuristics they use to find a maximum spanning tree in step 2. We give the pseudocode for this step in Algorithm 5.3, in which we use $\hat{F}_i \subseteq F_i$ to denote the set of functions for which a_i is sending messages in this step, and $F_i^p \subseteq F_i$ to denote the set of functions assigned to agent a_i by this step.

In more detail, this step involves linear communication and computation to greedily assign values to the variables representing each function. To this end, we initially assign a group of functions to each agent to send messages and compute for,[5] which we denote $\hat{F}_i \subseteq F_i$ — the decision as to which functions go in this set is arbitrary, so using a simple rule where the function goes to the agent with the lowest ID would work.

Once the $\hat{F}_i$ set has been established at agent a_i, the agent runs the procedure `initialise` (Algorithm 5.3, lines 1–7). First, the agent's assigned function set F_i^p is initialised to the empty set (line 2), and the agent's iteration count for each function, denoted $I(f_j)$, is initialised to 0 (line 3). Then, the agent sends `REQUEST`(f_j) messages requesting potential assignment values to each of the agents that can compute for each function in $\hat{F}_i$ (lines 4–6).

[5] We must stress that *linear* communication and computation is done in this step — the amounts are negligible compared to that of FMS or the rest of ST-DTDA.

Algorithm 5.3 `preprocess()` at agent a_i.

Require: $\hat{F}_i$

procedure `initialise`
$F_i^p = \emptyset$ `// initialise assigned functions`
$I(f_j) = 0$ for all $f_j \in \hat{F}_i$ `// initialise counters`
for $f_j \in \hat{F}_i$ **do**
  Send `REQUEST(`f_j`)` to all $a_k \in \mathcal{A}_j$ `// ask for values`
end for
end procedure

procedure `received REQUEST(`f_j`)` from agent a_k
$\Lambda_i = \chi_i(f_j) + \sum_{f_l \in F_i^p} \chi_i(f_l)$ `// compute potential assignment value`
Send $\langle \Lambda_i, f_j \rangle$ to a_k
end procedure

procedure `received` $\langle \Lambda_i, f_j \rangle$ from agent a_k
if Received from all $a \in \mathcal{A}_j$ **then**
  Send `ASSIGN(`f_j`)` to $\arg\min_{a_l \in \mathcal{A}_j} \Lambda_l$
  Send `UNASSIGN(`f_j`)` to agent f_j was previously assigned to
  $I(f_j) = I(f_j) + 1$
  if $I(f_j) < 10$ **then** `// Start next iteration`
    Send `REQUEST(`f_j`)` to all $a_k \in \mathcal{A}_j$
  end if
else
  Defer until next received message
end if
end procedure

procedure `received ASSIGN(`f_j`)` from agent a_k
$F_i^p = F_i^p \cup \{f_j\}$ `// add to assigned functions`
end procedure

procedure `received UNASSIGN(`f_j`)` from agent a_k
$F_i^p = F_i^p \setminus \{f_j\}$ `// remove from assigned functions`
end procedure

When an agent receives a `REQUEST`(f_j) message, it runs the procedure given in lines 8–11 in Algorithm 5.3. In more detail, the agent first computes Λ_i, which is an estimate of the potential assignment value at that agent, computed by summing the cost of computing function f_j ($\chi_i(f_j)$) with the cost incurred by all $f_l \in F_i^p$, i.e. the functions a_i is assigned to (line 9). This value is then sent back to the requesting agent (line 10).

Once an agent has received $\langle \Lambda_i, f_j \rangle$ messages from all agents which can compute for function f_j (line 13), the function computation for f_j is assigned to the agent with the smallest Λ_i by way of sending an `ASSIGN`(f_j) message (line 14) — if an agent were to have been assigned to f_j in a previous iteration of the algorithm, then an `UNASSIGN`(f_j) message is sent (line 15). Then, the iteration count for f_j is incremented (line 16), and, if the algorithm has not reached its final iteration (in this case we use 10), then the process begins again (lines 17–18).

Once all iterations have completed for all functions $f_j \in \mathcal{F}$, each agent has a set of functions they have been assigned to by the greedy algorithm, which they can use in step 2 as a starting point when building the spanning tree. We go into further detail on this next.

5.4.2 Step 2: Building the Spanning Tree

In this step, the agents build a spanning tree of their environment (the graph of agents and the functions they can compute, not the junction graph) in order to reduce the complexity of running DTDA over it. The key issue with this step is that producing any spanning tree of the environment requires ignoring some of the potential function-agent assignments in the environment. In so doing, we use the outcome of step 1 to produce an *approximation* of the environment, and run min-max on that approximation. Therefore, min-max is only able to select a solution from the set of possible solutions *contained in the spanning tree*, and so, it is important that the set of possible solutions contained in the spanning tree is of as high a quality as is possible — preferably containing the optimal solution, and if not, a solution with a makespan close to that of the optimal. Thus, to guide our search for an appropriate spanning tree, we use the greedy allocation found in step 1 to inform our approach.

In order to find an ST of the environment, we must first weight the edges of the graph. To this end, we give every edge in the graph a weight: $\omega_{ij} \in \mathbb{R}^+$, for every a_i and f_j, $a_i \in \mathcal{A}$ and $f_j \in F_i$. We then use this weight to find a spanning tree. Now, finding the right way to choose these weight values is a major challenge because selecting which edges should be in the spanning tree (and so, which agents should consider computing which functions) is a combinatorial problem which becomes more difficult as the number of edges in the graph increases. This is because removing the potential for an agent to compute a certain combination of functions could inadvertently cause a sub-optimal decision to be

made by another agent involved in one or more of those functions, combined with several others. Hence, we resort to a heuristic approach to choosing our weight values, whilst trying not to sacrifice solution quality, run-time, or bound tightness.

We propose three heuristics (H1, H2, and H3, detailed below) to weight the edges of the original factor graph, such that $\omega_{ij} = h(a_i, f_j)$, where $h : \mathcal{A} \times \mathcal{F} \rightarrow \mathbb{R}$. Next, we list our heuristics, and give some intuition behind them and which spanning tree needs to be found in each case, before describing example worst-case (in terms of solution quality) environments for each heuristic:

- **H1:** $h_1(a_i, f_j) = \chi_i(f_j) + \sum_{f_k \in F_i^p, k \neq j} \chi_i(f_k)$. This is the simplest, most obvious heuristic of the three, simply summing the time it takes agent a_i to compute function f_j with the total time it would take agent a_i to compute for all functions in F_i^p (the greedy assignment found in step 1) to weight edges, and then finding a minimum spanning tree to remove higher-weight edges (and so, high-cost agent-function assignments). The intuition behind this heuristic is that removing high-weight edges means removing the most costly agent-function assignments, which are the least likely to be in the optimal solution.

- **H2:** $h_2(a_i, f_j) = \chi_i(f_j) + \sum_{f_k \in F_i^{st} \cup F_i^p, k \neq j} \chi_i(f_k)$, where $F_i^{st} \subseteq F_i$ denotes the set of edges which have so far been selected for the spanning tree (i.e., this set is incrementally updated as agents select spanning tree edges). In more detail, this heuristic requires the weights of the edges incident on agent a_i to be recalculated every time an edge incident on a_i is selected to be in the spanning tree, and sums this with the total time it would take agent a_i to compute for all functions in F_i^p (the greedy assignment found in step 1). The intuition behind this heuristic comes from the fact that most spanning tree algorithms generally operate on the principle of building the spanning tree by beginning from individual nodes and selecting edges to join them, rather than pruning edges from the graph to make a spanning tree. Generally, spanning tree algorithms such as GHS join nodes on their lowest-weight edges, one by one, until the tree is complete. Thus, once an agent has made a spanning tree edge between itself and a function, it can recalculate the edge weights for all other edges it is incident on. This, then, tries to find an optimal spanning tree by directly minimising the worst-case makespan at each agent, where the worst-case makespan is where the agent performs every function it is connected to in the spanning tree. Thus, as with $H1$, here, we find a *minimum* ST in order to remove higher-weight edges, which in this case are incident on slower agents, or agents that already are highly connected in the spanning tree. However, the key difference between $H2$ and $H1$ is that $H2$ is *iterative*, updating the edge weights as the spanning tree is built.

- **H3:** $h_3(a_i, f_j) = \min_{a_k \in \mathcal{A}_j, k \neq i} \chi_k(f_j) + \sum_{f_l \in F_i^p, l \neq j} \chi_k(f_l)$. Using this heuristic to find a *maximum* ST aims to minimise the worst-case error bound obtained in step

4 (Equation 5.9), by focussing on which edges we wish to *keep* in the spanning tree, as opposed to the edges that we wish to remove. This is because this worst-case error bound is computed by running min-max over all variables whose domains have been changed, so by minimising the potential makespans of variables whose domains have changed, we minimise the error bound. In order to do this, we must ensure that the ST edges from each agent connect to functions with minimal cost. Thus, when computing the bound as the makespan over all functions which have had edges removed (as we explain in step 4), the bound will be low. We are aware that this heuristic could produce a lot of equal weight edges, as the minimisation here will lead to many of the edges from an agent being the same weight. Therefore, in cases where two edge weights are equal, we consider the edge with the highest $\chi_i(f_j)$ value to be the heaviest.

Having chosen a heuristic to use to weight the edges of the graph of agents and the functions they can complete, we then use the edge weights to find an ST that minimises/maximises the weight. As mentioned above, we find the minimum ST for $H1$ and $H2$, and the maximum ST for $H3$. To find an ST, we use iGHS (see Section 4.4.1)). As with BFMS, we recognise that the use of iGHS introduces additional communication/-computation cost; however, iGHS is efficient in both aspects. Thus, we believe that the additional communication and computation required by this step is justified given the resulting reduction in communication and computation needed by min-max.

Once we have found the ST, we formally express the state of the environment as follows: we define the set $F^c \subseteq F$ to contain the functions whose domains have been modified in the ST (i.e., at least one edge has been cut between a function f_i and an agent a_j). Similarly, we define $F^t \subseteq \mathcal{F}$ has the functions whose domains haven't been modified in the ST. Thus, $F^c \cup F^t = \mathcal{F}$ and $F^c \cap F^t = \emptyset$. Given this, the state at each agent is as follows: for each agent a_i, we have $F_i^c \cup F_i^t = F_i$, where F_i^c is the set of functions a_i is not connected to in the ST, and F_i^t is the set of functions a_i is connected to in the ST.

The junction graph that results from the tree-structured problem always corresponds to an acyclic constraint network, upon which GDL algorithms are guaranteed to converge to the optimal solution (Aji and McEliece, 2000). Observe that the junction graph is a dual graph, as the nodes in the junction graph represent agents, that stand for constraints, and the edges represent common functions, two agents connected through the same variable. As a result of building the ST, any pair of agents in the environment are connected through a single path, and so, the dual representation of the modified environment is acyclic and therefore the junction graph is acyclic too.

5.4.3 Step 3: Applying Min-max

Now that we have the ST representation of the function allocation problem, we run min-max on the junction graph representation of the ST. $\tilde{Y}$ contains all problem variables, including those with modified domains (i.e., with some elements removed as a result of finding the ST). The junction graph is built in almost the same way as detailed in Section 3.5.3, but only the variables representing the functions in F_i^t are contained in agent a_i's clique. Similarly, we modify the domain of each variable y_k, to give a modified variable $\tilde{y}_k$ with domain $\tilde{\Delta}_k$, such that $\tilde{\Delta}_k = \{i | a_i \in \Delta_k \vee f_k \in F_i^t\}$. Thus, clique a_i contains the set of variables which we denote $\tilde{Y}_i = \{\tilde{y}_k | f_k \in F_i^t\}$. Additionally, clique a_i is only connected to another clique a_j if $\tilde{Y}_i \cap \tilde{Y}_j \neq \emptyset$.

In the objective function, we minimise the makespan incurred from functions in F_i^c at each agent. Hence, formally, a_i's potential function is:

$$\psi_i(\tilde{\mathbf{Y}}_i) = \sum_{\tilde{y}_k \in \tilde{\mathbf{Y}}_i, \tilde{y}_k = i} \chi_i(f_k) \tag{5.8}$$

Given this, we can proceed with min-max as described in Section 5.3.1, simply substituting $\tilde{Y}_i$ wherever Y_i is used, and $\tilde{y}$ wherever y is used.

In the next, and final, step, we show how we can bound the quality of the solutions obtained by ST-DTDA. In what follows, we define a bound on the absolute error of the approximate solution, and then show how we use this, and a lower bound, to estimate the approximation ratio of our solution. As before, Y contains the problem variables, with their original domains. Y^c contains all variables y_i corresponding to the functions in F^c, whose domains have been modified, or, more formally, $Y^c = \{y_i | f_i \in F^c\}$. Similarly, Y^t contains all variables y_i corresponding to the functions in F^t, whose domains have not been modified, or, formally, $Y^t = \{y_i | f_i \in F^t\}$.

5.4.4 Step 4: Computing the Approximation Ratio

The final step of ST-DTDA is for the agents to compute a worst-case bound on the quality of the solution found. Such worst case bounds provide an indication of the quality of the spanning tree found in step 2, which we denote as $\tilde{V}^*$, and we denote the value of the minimum makespan for the variables in Y^c with $\tilde{V}^{c,*}$. $\tilde{V}^{c,*}$ is defined as follows:

$$\tilde{V}^{c,*} = \min_{\tilde{\mathbf{Y}}^c} \max_{a_i \in \mathcal{A}} \psi_i(\tilde{\mathbf{Y}}_i^c) \tag{5.9}$$

As we formally prove in Section 5.4.6, $\tilde{V}^{c,*}$ bounds the distance of the approximate solution $\tilde{Y}^*$ we obtained for the tree-structured approximate problem with respect to

the optimal solution of the original problem, which we denote V^*. Thus:

$$\tilde{V}^{c,*} \geq \tilde{V}^* - V^* \tag{5.10}$$

Therefore, to compute $\tilde{V}^{c,*}$, we must find:

$$\tilde{\mathbf{Y}}^{c,*} = \arg\min_{\tilde{\mathbf{Y}}^c} \max_{a_i \in \mathcal{A}} \sum_{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,\prime} | \tilde{y}_k = i} \chi_i(f_k)$$

which we can do by applying min-max using only the variables in Y^c.

To do this, min-max is run in the same way as described in Section 5.2, but using only the variables in Y^c. At the completion of min-max, every agent is aware of the configuration $\tilde{\mathbf{Y}}^{c,*}$, and so, the value of the configuration, which we denote $\tilde{V}^{c,*}$. Now, whilst this error bound is tight, there can be cases where the makespan of $\tilde{\mathbf{Y}}^*$ (which we denote $\tilde{V}^*$) is equal to $\tilde{V}^{c,*}$. In such cases, there exists no error bound, and so, using this bound alone the agents are not guaranteed to be able to assess the quality of the solution they find. To combat this, we compute a lower bound on the optimal makespan, specifically:

$$V^{LB} = \max_{f_k \in \mathcal{F}} \min_{a_i \in \mathcal{A}} \chi_i(f_k) \tag{5.11}$$

Put simply, V^{LB} is the time taken for the fastest agent to perform the longest function, and therefore an estimate of the smallest makespan achievable. This value is computed amongst the agents through a value propagation phase.

Given the absolute error bound $V^{c,*}$ and V^{LB}, we estimate the approximation ratio ρ of any approximate solution as follows:

$$\rho = \frac{\tilde{V}^*}{\max\left(\tilde{V}^* - \tilde{V}^{c,*}, V^{LB}\right)} \tag{5.12}$$

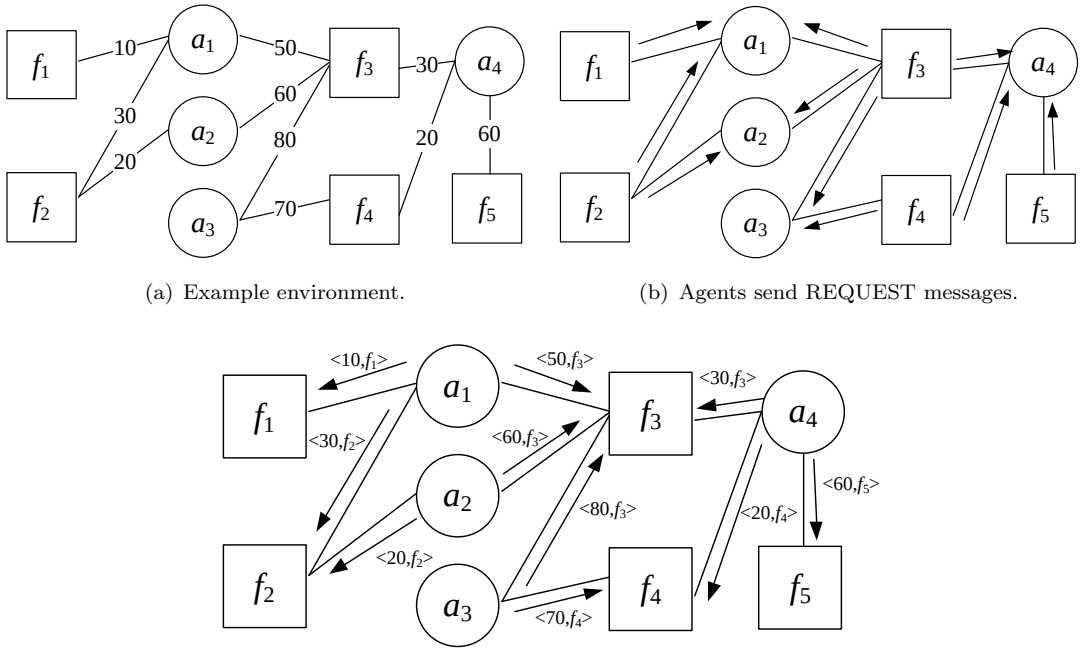
This approximation ratio allows us to estimate the quality of the solution computed, such that:

$$V^* \leq \frac{\tilde{V}}{\rho} \tag{5.13}$$

In doing this, we ensure that agents running ST-DTDA can always compute a bound on the quality of the solution they have computed. In the next section, we provide a worked example of ST-DTDA, before going on to prove some key properties of the algorithm.

5.4.5 Worked Example

Figure 5.2 (a) shows the environment upon which we run this worked example. In Figures 5.2 (b) and (c), we show a single iteration of the first step of ST-DTDA: preprocessing. In more detail, the agents begin by sending a `REQUEST` message, effectively from each



(a) Example environment.

(b) Agents send REQUEST messages.

(c) Functions respond with value messages.

FIGURE 5.2: Worked example of the operation of ST-DTDA, part 1.

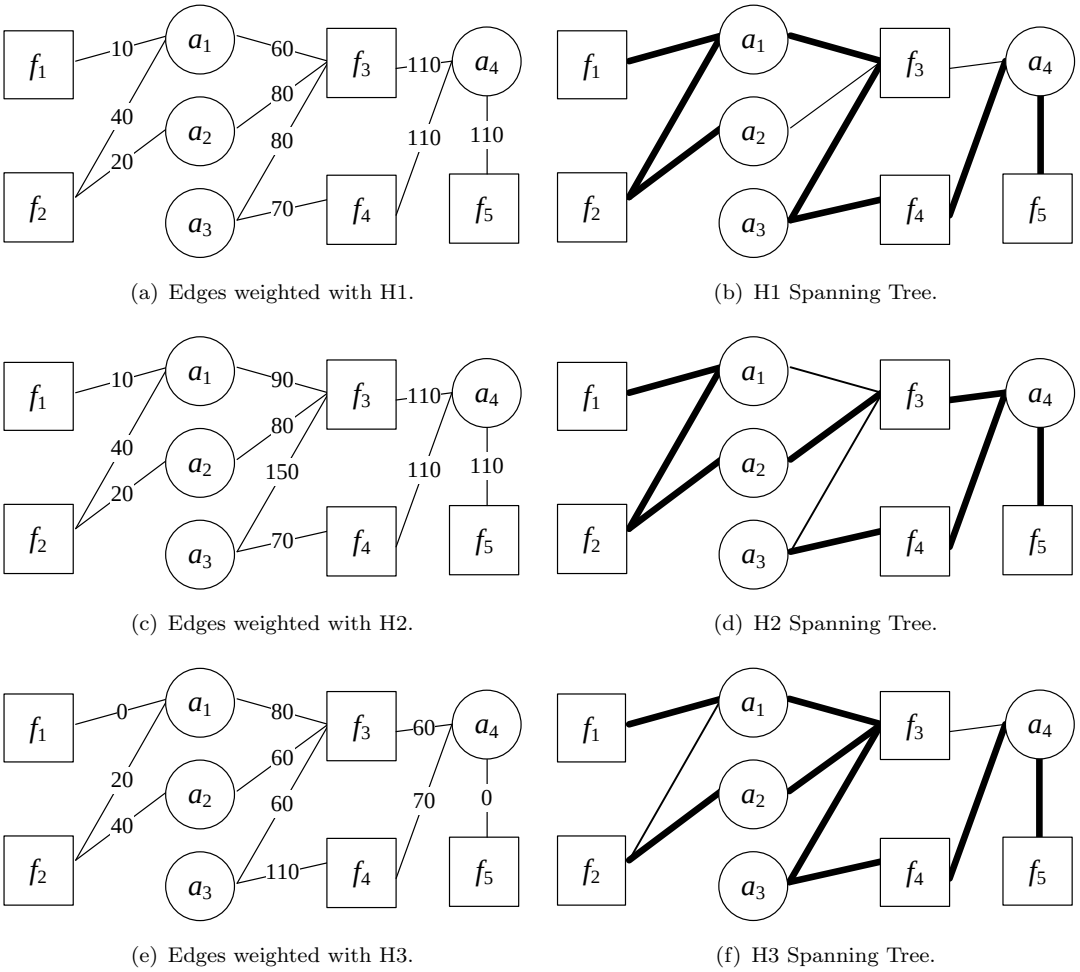
function to each agent that can compute it (as indicated by the arrows in Figure 5.2 (b)). On receipt of a `REQUEST`(f_j) from another agent, each agent computes its Λ_i value as given in Algorithm 5.3, and sends that back to the agent communicating for f_j (this is denoted by the text on the arrows in Figure 5.2 (c)). This completes one iteration of preprocessing — 9 further steps would normally be carried out after this, but for the sake of brevity, we only show one here.

The outcome of the preprocessing phase is the definition of F_i^p sets for each agent. More specifically, after the preprocessing step, a_1 is assigned f_1, so $F_1^p = \{f_1\}$, and, in a similar vein, $F_2^p = \{f_2\}$, $F_3^p = \emptyset$, and $F_4^p = \{f_3, f_4, f_5\}$. The makespan of this example is $\max(10, 20, 110) = 110$, which is suboptimal since the optimal makespan is 70.

In step 2, we use our heuristics to find a spanning tree. We explain how this is done next:

$H1$ — Figure 5.3 (a) shows the graph labelled with the edge weights computed using $H1$: for example, the edge between a_1 and f_2 has weight $\omega_{12} = 40$ because $\chi_1(f_2) + \sum_{f_j \in F_1^p} \chi_1(f_j) = 30 + 10$. The minimum spanning tree of this graph is given in Figure 5.3 (b), where the edge between a_1 and f_2 is omitted from the graph to form the spanning tree.

$H2$ — $H2$ involves updating edge weights as the spanning tree algorithm operates, so the weights labelling the edges in Figure 5.3 (c) denote the final edge weights at the completion of the MST algorithm. Broadly speaking, the spanning tree algorithm



(a) Edges weighted with H1.

(b) H1 Spanning Tree.

(c) Edges weighted with H2.

(d) H2 Spanning Tree.

(e) Edges weighted with H3.

(f) H3 Spanning Tree.

FIGURE 5.3: Worked example of the operation of ST-DTDA, part 2.

will proceed as follows: when all weights are initially computed as in $H1$, the lowest-weight edge is a_1 to f_1, so this edge will be added to the spanning tree, and the other edge weights for a_1 will be updated. The next edge to be added is a_2 to f_2, then a_1 to f_2, then a_3 to f_4, then a_2 to f_3, a_4 to f_3, a_4 to f_4, and finally, a_4 to f_5. Thus, the spanning tree produced at the end of this is given in Figure 5.3(d).

$H3$ — The edge weights produced by $H3$ are given in Figure 5.3(e), in which the edges between f_1 and a_1, and f_5 and a_4, are of weight 0 since no other agents can compute for those functions. Recall that for $H3$ we find a *maximum* spanning tree, not a minimum one, and so the spanning tree found using $H3$ is given in Figure 5.3(f).

Now, to illustrate step 3, consider the spanning tree found by H1 in Figure 5.3(b). After a junction tree has been built, and min-max run on the junction tree, as described earlier in Section 5.3.4, min-max will find the optimal solution where $y_1 = a_1$, $y_2 = a_2$, $y_3 = a_1$, $y_4 = a_3$, and $y_5 = a_4$, and the makespan is $\max(10 + 50, 20, 70, 60) = 70$. Thus, the solution value found in this step, denoted $\tilde{V}^*$, is 70.

Next, to compute the error bound in step 4, consider the spanning tree given in Figure

5.3(b), which is the spanning tree produced as a result of using heuristic H1 in this example. The only edges removed are (a_2, f_3) and (a_4, f_3), so in this case, $Y^c = \{y_3\}$ to represent f_3, and so, min-max would be run amongst the agents to find the optimal value of y_3 (which is our error bound), which, in this case, is a_1, since $\arg_{a_1,a_3} \min(50, 80) = a_1$. Thus, $\tilde{V}^{c,*} = 50$ in our example, which means that any solution found in step 3 will be at most 50 away from the optimal solution value.

Finally, to compute the approximation ratio, V^{LB} is computed as in Equation 5.11, as follows:

$$\begin{aligned} V^{LB} &= \max\left(\min_{i\in\{1\}}(\chi_i(f_1)), \min_{i\in\{1,2\}}(\chi_i(f_2)), \min_{i\in\{1,2,3,4\}}(\chi_i(f_3)), \min_{i\in\{3,4\}}(\chi_i(f_4)), \min_{i\in\{4\}}(\chi_i(f_5))\right) \\ &= \max\left(\min(10), \min(20, 30), \min(50, 60, 80, 30), \min(70, 20), \min(60)\right) = 60 \end{aligned}$$

Given $\tilde{V}^{c,*} = 50$, $\tilde{V}^* = 70$, and $V^{LB} = 60$, we compute the approximation ratio, according to Equation 5.12, as follows:

$$\rho \approx \frac{70}{\max\left(70 - 50, 60\right)} \approx \frac{70}{60} \approx 1.16$$

Thus, in this case, H1 achieves an approximation ratio of 1.16, despite the fact that the approximate solution is equal to the optimal.

5.4.6 Properties of ST-DTDA

First, we prove the bounded approximation provided by ST-DTDA — specifically, we must prove that Equation 5.10 holds. Since $\rho = \frac{\tilde{V}^* - \tilde{V}^{c,*}}{\tilde{V}^*}$ and $V^{c,*} \leq V^{c,'}$, $\forall \mathbf{Y}^{c,'} \in \tilde{\mathbf{Y}}^c$, in order to show that Equation 5.10 holds, it is sufficient to prove the following theorem:

Theorem 5.7 (Bounded Approximation)**.** $\forall \tilde{\mathbf{Y}}^{c,'} \in \tilde{\mathbf{Y}}^c$, $V^{c,'} \geq \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i^{t,*}; \tilde{\mathbf{Y}}_i^{c,'}) - V^*$

Proof. Since the value of the optimal configuration V^* is guaranteed to be lower than the value of any other configuration that we can build by varying the configuration of variables in Y^c with respect to the optimal, the following equation holds for all $\tilde{\mathbf{Y}}^{c,'} \in \tilde{\mathbf{Y}}^c$:

$$\begin{gathered} \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i^*) \leq \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i^{t,*}; \tilde{\mathbf{Y}}_i^{c,*}) = \\ \max_{a_i \in \mathcal{A}} \left(\sum_{\substack{y_k^* \in \mathbf{Y}^{t,*} \\ y_k^* = i}} \chi_i(f_k) + \sum_{\substack{y_k^* \in \mathbf{Y}^{c,*} \\ y_k^* = i}} \chi_i(f_k) \right) \leq \max_{a_i \in \mathcal{A}} \left(\sum_{\substack{y_k^* \in \mathbf{Y}^{t,*} \\ y_k^* = i}} \chi_i(f_k) + \sum_{\substack{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,'} \\ \tilde{y}_k = i}} \chi_i(f_k) \right) \end{gathered} \tag{5.14}$$

Also, since the difference between the value of a configuration with respect to the value of the optimal must be lower than the maximum difference between the agent's computation times in both configurations, the following equation holds:

$$\max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i^{t,*}; \tilde{\mathbf{Y}}_i^{c,*}) - \max_{a_i \in \mathcal{A}} \psi_i(\mathbf{Y}_i^*) \leq \max_{a_i \in \mathcal{A}} \left(\sum_{\substack{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,'} \\ \tilde{y}_k = i}} \chi_i(f_k) - \sum_{\substack{y_k \in \mathbf{Y}^{c,*} \\ y_k = i}} \chi_i(f_k) \right)$$

and, $\forall a_j \in \mathcal{A}$:

$$\max_{a_i \in A} \sum_{\substack{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,'} \\ \tilde{y}_k = i}} \chi_i(f_k) - \sum_{\substack{x_k \in \mathbf{Y}^{c,*} \\ y_k = j}} \chi_i(f_k) \leq \max_{a_i \in \mathcal{A}} \sum_{\substack{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,'} \\ \tilde{y}_k = j}} \chi_i(f_k)$$

Thus, $\max_{a_i \in \mathcal{A}} \sum_{\substack{\tilde{y}_k \in \tilde{\mathbf{Y}}^{c,'} \\ \tilde{y}_k = j}} \chi_i(f_k)$ is an absolute bound on the distance between $\tilde{V}$ and V^* for any $\tilde{\mathbf{Y}}^{c,'} \in \tilde{\mathbf{Y}}^c$ and in particular for $\tilde{Y}^{c,}$. □

Theorem 5.7 states that for each possible assignment of the modified variables (i.e., the variables in $\tilde{V}^{c,*}$), the difference between the value of the approximate solution (optimal on the ST), $\tilde{V}^*$, and the value of the makespan when only considering the modified variables in the ST, $\tilde{V}^{c,*}$, is always lower than the value the optimal solution of the original problem, V^* (i.e. it is a lower bound).

We also find that ST-DTDA is superstabilizing subject to the predicates given in Section 5.3.5, since the only significant computational change we made to DTDA here is the introduction of message passing phases (steps 1 and 4) which can be re-run from scratch without violating either of the predicates. Additionally, finding the spanning tree in step 2 can be done by iGHS, which was shown to be superstabilizing in Section 4.4.1.6. Finally, steps 3 and 4, which use min-max, can use storage at each agent and decision rules at each agent, as discussed in Section 5.3.3, in order to reduce communication and computation overheads after a change, and to ensure that the algorithm is superstabilizing.

Furthermore, ST-DTDA is also an anytime algorithm, since the algorithm can be stopped at any point and a valid solution obtained by each agent using marginal contribution values, as in DTDA.

5.4.7 Summary

As we mentioned in Section 5.3.6, DTDA is intractable, since the μ messages are exponential in both size and complexity. Now, since ST-DTDA always runs min-max over a junction tree, and GDL algorithms have polynomial complexity when run over

junction trees (Aji and McEliece, 2000), we know that the computational complexity of ST-DTDA is polynomial, too.

To relate this back to the aims and objectives highlighted in Section 1.3, this algorithm will allow us to fully decentralise the computation of our FMS algorithm and its variants in two ways. Firstly, ST-DTDA finds an efficient distribution of the computation of factors amongst agents, and secondly, if an agent were to be removed from the environment, another agent could take over the computation it was performing without significant disruption.

Given all this, in the next section, we empirically evaluate ST-DTDA in three ways: First, we show the solution quality achieved by ST-DTDA relative to some standard benchmarks. Second, we show the accuracy of the estimated approximation ratio computed in step 5 of ST-DTDA compared to the actual solution quality, so that we can assess how tight the bound we find is. Finally, we compare the scalability (in functions and agents) of ST-DTDA.

5.5 Empirical Evaluation

In this section, we compare the approximate solutions found by ST-DTDA and a number of other algorithms, thus establishing the first benchmarks for decentralised solutions to $R||C_{\max}$. Namely, we compare ST-DTDA (using heuristics $H1$, $H2$, and $H3$ as given in Section 5.4.2), DTDA, an optimal centralised algorithm (in order to assess how far our solutions are from the optimal), and a DSA-based greedy algorithm (to show the benefit of combining both this algorithm and min-max). In more detail, the optimal centralised algorithm (CA) operates by solving a mixed integer program to find the optimal solution. We formulate the problem as a binary integer program, and then use IBM ILOG CPLEX 12.3[6] to find an optimal solution assigning functions to agents. Next, our DSA-based greedy algorithm (Greedy) simply assigns functions to agents based on the outcome of step 1 of ST-DTDA (see Section 5.4.1). In both these cases, we consider exactly the same problem that ST-DTDA does — i.e., each agent can only compute a subset of the functions.[7]

In all our experiments we used 500 random graphs, and 500 scale-free graphs, with each of agents and tasks as hubs.[8] Given this, we evaluate our algorithm by varying three key parameters: function to agent ratio ($\frac{|\mathcal{F}|}{|\mathcal{A}|}$), cost value variance ($\sigma^2$), and mean agent connectivity (δ_a), since these are the three factors which are most likely to impact solution quality, and are settings varied by other work in the area (for example, (Wotzlaw, 2006)). We describe the settings specific to these parameters below.

[6] See `www.ibm.com/software/integration/optimization/cplex-optimizer/`

[7] Note that we do not compare to any other existing approximate algorithms for $R||C_{\max}$ because, as we said earlier (Section 2.4.2), there exist *no* decentralised algorithms for $R||C_{\max}$.

[8] The graphs are generated as described in Section 4.5.

1. **S1 — Function to agent ratio:** We vary this to assess how our algorithm performs as the scale of the environment increases. $|\mathcal{A}| = 40$ and $|\mathcal{F}| \in \{20, 40, \ldots, 100\}$. We do not need to vary the number of agents here as the complexity of a problem instance comes from agents being able to perform many functions — increasing the number of agents makes finding a solution easier because agents are less likely to need to compute combinations of functions. Random graphs were generated with $\delta_a = 3$, and v_{ij} was drawn from a Poisson distribution with a variance (and so, mean) of $\sigma^2 = 100$.

2. **S2 — Cost value variance:** We vary this to see if the diversity of cost values has an impact on the solution quality and bound tightness obtained. Here, $|\mathcal{A}| = 20$ and $|\mathcal{F}| = 40$. v_{ij} was drawn from a Poisson distribution with a variance (and so, mean) of $\sigma^2 \in \{1, 10, 20, \ldots, 100\}$. Random graphs were generated with $\delta_a = 3$, meaning each agent can compute for an average of 3 functions.

3. **S3 — Link density:** Varying link density allows us to assess the performance of our algorithm under differing levels of sparsity in agent-function interactions. Here, $|\mathcal{A}| = 20$ and $|\mathcal{F}| = 40$. v_{ij} was drawn from a Poisson distribution with a variance (and so, mean) of $\sigma^2 = 100$. Graphs were generated with $\delta_a \in \{2, 3, 4, 5, 6\}$.

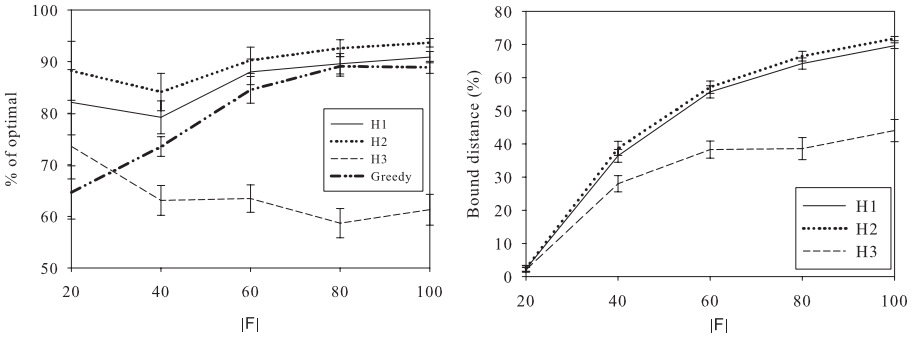
In doing these experiments, we evaluate the performance of the algorithms using two metrics: solution quality and bound tightness (both of which are paralleled by the makespan deviation metric used by (Wotzlaw, 2006)). We explain these metrics below:

1. **Solution quality:** we assess the quality of solutions obtained by each algorithm by dividing the achieved makespan by the optimal makespan (i.e., that obtained by CA), over graphs of various topologies and densities (defined below), and use this to plot the percentage of the optimal obtained.

2. **Bound tightness:** to assess bound tightness, we multiply the approximation ratio obtained in step 4 of ST-DTDA (see Section 5.4.4) by the optimal solution obtained by CA, and plot the distance between that and the actual percentage of the optimal achieved by each of the three heuristics.

Each of these metrics are plotted with 95% confidence intervals as error bars. Against this background, in the remainder of this section, we evaluate the performance of our algorithms under varying agent to function ratio, cost value variance, and link density, in terms of solution quality, run-time, and bound tightness.[9]

[9]Note that we only reproduce results for random graphs here because we found there to be no significant difference in performance between our results on scale-free and random graphs.



(a) Solution quality

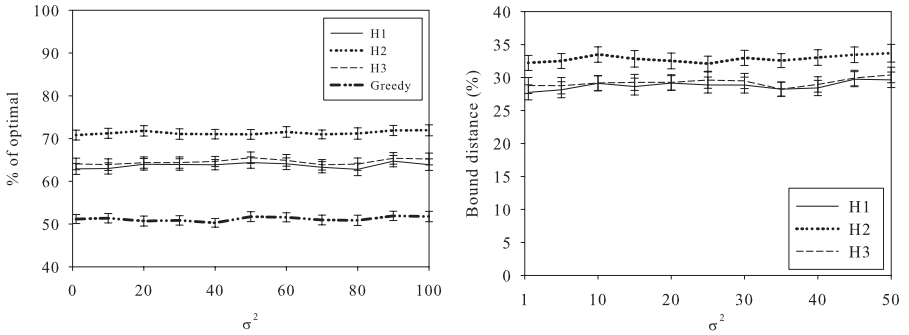
(b) Bound tightness

FIGURE 5.4: Solution quality achieved and bound tightness against increasing $|\mathcal{F}|$, and so, increasing function to agent ratio.

5.5.1 Function to Agent Ratio

Figure 5.4(a) depicts the solution quality achieved by $H1$, $H2$, and $H3$, as compared to Greedy on random graphs with $\delta_a = 3$ using settings S1. The general upward trend (for H1, H2 and greedy) visible in the graph shows how the $R||C_{\max}$ problem actually gets *easier* to solve as the function to agent ratio (i.e., $\frac{|\mathcal{F}|}{|\mathcal{A}|}$) increases. This is because, as the average number of functions each agent will perform increases, the impact that one function being assigned to a slower agent has on the makespan decreases. ST-DTDA with H3 is the only algorithm to actually get *worse* as the number of functions increases — this is because the potential for H3 to find the wrong solution when concentrating on improving the bound increases as the number of functions that could be incorrectly assigned as a result increases. Nevertheless, H1 and H2 outperform Greedy by 1–27% (H1) and 3–36% (H2), with the biggest differences being at smaller values of $|\mathcal{F}|$, where greedy finds poorer quality solutions due to the potential for a single wrong variable assignment to significantly impact the overall makespan being bigger on smaller graphs. As expected, H2 obtains the best solution quality overall due to the recomputation performed while forming its ST (see Section 5.4.2 for details).

In contrast to the solution quality results in Figure 5.4(a), observe the upward trend in Figure 5.4(b) (where lower numbers are better) — the bounds found by our heuristics get *looser* as the number of functions in the environment increases. This is because, as the number of functions increases, the number of edges that can potentially be pruned to form the spanning tree increases. Thus, it becomes more difficult to find an accurate bound in step 4 of ST-DTDA. Here, as expected, we see that H3 obtains a bound that is 16-66% tighter relative to H1, and 22-72% tighter relative to H2, thus showing that H3 is the best choice of the three heuristics if bound tightness is an issue.



(a) Solution quality (b) Bound tightness

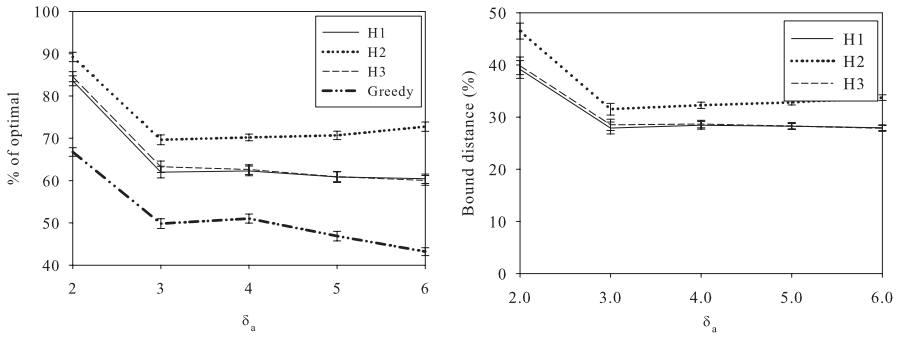
FIGURE 5.5: Solution quality achieved and bound tightness against increasing σ^2.

5.5.2 Cost Value Variance

In Figure 5.5 we show that the solution quality and bound tightness achieved by our heuristics under settings S2 are not significantly impacted by increasing cost value variance, since all three heuristics achieve similar performance over all examined values of σ^2. As predicted, in Figure 5.5(a), H2 obtains the best solutions here (37–41% improvement relative to greedy), but, in Figure 5.5(b), produces looser bounds (around 15% looser relative to H3). Interestingly, in this experiment, H1 and H3 achieved around the same solution quality (22–27% improvement relative to greedy), and around the same bound tightness. This is because, in environments with low function to agent ratios like this one, H3 performs very similarly to H1. This happens when an agent can perform fewer functions, since its edge weightings will be very similar to those in H1 because the minimisation part of the heuristic is unlikely to have an impact.

5.5.3 Link Density

We found that, when using settings S3, the solution quality and bound tightness achieved by our heuristics does not vary significantly with link density where link density is greater than 2. In more detail, Figure 5.6(a) shows that the quality of solutions obtained by H2 is at most 68% higher relative to the greedy algorithm, and up to 30% higher relative to H1 and H3. The difference between H2 and H1/H3 increases with δ_a, as the solution qualities achieved by H1 degrade faster with δ_a than those found by $H2$. This is because, as discussed in Section 5.4.2, whilst both heuristics are myopic (i.e., only take one agent into consideration), when finding the ST, $H2$ takes into account the fact that the overall makespan comes from the agent with the longest completion time, and so weights the edges such that agents which could incur large makespans are less likely to, as discussed in Section 5.4.2. In contrast, $H1$ weights the edges such that a slower agent is less likely to be chosen to compute a costly function, regardless of the combination of other functions that agent is likely to compute for. Thus, as the amount of functions an



(a) Solution quality (b) Bound tightness

FIGURE 5.6: Solution quality achieved and bound tightness against increasing δ_a.

agent can compute (σ_a) increases, the myopic nature of $H1$ becomes detrimental to the solution quality attained.

In terms of bound tightness, we see in Figure 5.6(b) that H2 achieves a 10–21% looser bound than H1 and H3 — because H2 does not specifically aim to find a tight bound, instead aiming to find better quality solutions. Furthermore, we found that H1 and H3 behave very similarly to each other (obtaining solution quality and bound tightness within 1% of each other) for all graph densities tested, which only goes to further prove H3 and H1 perform very similarly on small graphs. An interesting result shown in Figure 5.6 (b) is that, beyond $\delta_a = 3$ the quality of the bounds found by the heuristics actually *improves* slightly with increasing δ_a. This is because as δ_a increases, the number of edges pruned to create an ST increases, which increases the size of the space from which the absolute error bound is found in step 3 (see Section 5.4.3), thus allowing it to be more accurate. However, whilst the bound might be tighter with H1 and H3, the overall solution quality is higher with H2, which highlights the tradeoff between solution quality and bound tightness introduced by our heuristics. This tradeoff is backed up by the fact that, throughout our experiments, while H3 clearly achieves poorer results with respect to solution quality, it actually consistently achieves tighter bounds than H2.

5.6 Summary

We can summarise our results as follows. Specifically, we have presented the first decentralised algorithm for the scheduling on unrelated parallel machines problem, known as $R||C_{\max}$. In so doing, we have advanced toward better meeting the requirements defined in Section 1.3 (particularly Requirements 1, 2 and 6) by greatly improving the scalability of the vanilla min-max algorithm, providing a decentralised, robust computation distribution for use in conjunction with FMS, and by providing bounds on the quality of the computation distribution produced. In more detail, DTDA uses the min-max localised message-passing algorithm to compute an approximate solution to the compu-

tation distribution problem in a completely decentralised manner, whilst also providing a worst-case per-instance bound on solution quality. Despite this, DTDA involves exponential computational complexity at each agent when computing a message to send, and so, as we showed in our evaluation, is intractable even on relatively small graphs.

To combat this, we presented ST-DTDA, an extension to DTDA which uses maximum/minimum spanning trees to run the min-max algorithm on a simplified approximation of a given problem, in order to produce good quality solutions in much less time, again with a worst-case per-instance bound on solution quality. Now, in order to find an appropriate ST to run the algorithm on, we proposed and evaluated three admissible heuristics for weighting the importance of links between agents and functions they can do. In so doing, we have shown how the key challenge for improving ST-DTDA is found in choosing a heuristic which finds the right balance between solution quality and bound tightness. In particular, the heuristics we use show that good quality solutions can be found using a myopic heuristic (i.e., not taking into account other agents or which functions the other agents can perform), but also show that there is some merit in considering the wider impact of decisions made: a purely myopic heuristic (H1) did not produce as good quality solutions as one which took a worst-case estimate of the overall solution value into account (H2). However, whilst this is true, we found that solution quality bounds found when using the myopic heuristic (H1) were, for the most part, up to 21% tighter than those found using H2. In addition to this, H3 was created specifically to tighten the bounds, and was shown to achieve bounds up to 66% tighter relative to H1 and H2, in the best case — however, this was at the expense of solution quality, under which metric H3 can perform worse than greedy (in environments with more than 30 functions, as shown under experiment settings S1). This helps to explicitly highlight the trade-off that exists between solution quality and bound tightness: H2 would be the clear choice if solution quality were a concern, however, if the concern is to have a very good estimate of how good the solution is, without minding how good the solution is, then H3 would be the best choice.

With all this in mind, we can go back to the requirements, outlined in Section 1.3 and assess progress. Specifically:

1. **Scalability** — As shown earlier, DTDA does not scale well in the number of agents and functions. However, the introduction of preprocessing in ST-DTDA allows the algorithm to scale to hundreds of agents with very little (linear) required storage.

2. **Robustness** — As in FMS and its extensions, the decision making in ST-DTDA is spread throughout all agents in the system. Thus, the removal of one or more agents will not cause the algorithm to be completely re-run.

3. **Efficiency** — In ST-DTDA, we use decision rules at each agent to minimise the number of extraneous messages sent after changes in the environment. Thus, whilst

further steps could be taken to further increase the efficiency of the communication used by ST-DTDA, we have gone some way to address this requirement here, especially when taking into account the lack of alternative decentralised algorithms for computation distribution (see Section 2.4.2).

4. **Adaptiveness** — ST-DTDA will recover more efficiently from changes in the environment than simply using min-max alone, due partly to the preprocessing step significantly reducing computation and communication, and partly due to the decision rules at each agent. Thus, in so doing, ST-DTDA specifically caters for a dynamic environment.

5. **Quality** — Here, we have shown that ST-DTDA generally provides solutions of good quality — especially when using heuristic H2, and so, we consider this requirement satisfied.

6. **Boundedness** — ST-DTDA can always provide worst-case bounds on the quality of the solutions it produces on arbitrary graphs, and, being a GDL algorithm, is proven to always find the optimal solution on a tree-structured graph. In addition to this, we have shown that there exists a tradeoff between solution quality and bound quality, introduced by the need for heuristics when finding the MST.

	ST-DTDA
Scalability	✓✓
Robustness	✓✓
Efficiency	✓
Adaptiveness	✓
Quality	✓
Boundedness	✓✓

TABLE 5.1: Outline of how our contributions in this chapter relate to the requirements defined in Section 1.3. Here, we denote a requirement not satisfied by ×, and a requirement (strongly) satisfied by ✓(✓✓).

In Table 5.1, we provide a summary of how ST-DTDA meets our requirements. In more detail, ST-DTDA meets all six of the requirements, as described above. As explained in Chapter 4, for FMS and its extensions to be truly decentralised, and so, robust to agent failure (requirement 2), the computation of functions must be allocated to agents. Without this distribution of function computation, FMS cannot be easily decentralised, since the computation of function nodes has to be done somewhere. Moreover, this allocation of functions to agents must be done in such a way that the time it takes to run FMS is kept to a minimum — ensuring that agents with lower computational power do not end up computing complex functions over agents that could compute the functions more quickly. In this chapter, we have presented our approach to solving this problem, and shown empirically that our approach finds solutions with makespans that are at most 68% lower relative to using a simplistic greedy approach — this translates to finding a distribution of computation which ensures that the execution of FMS takes

place up to 68% faster relative to a simplistic greedy algorithm. Despite this, we should stress that ST-DTDA marks initial work into the space of decentralised solutions to the unrelated parallel machines problem. In this chapter we have found that ST-DTDA performs particularly well on small-scale environments, where there are few agents and tasks, and the links between them are sparse. Thus, the direction for future work in this space should be to try and find an algorithm which scales better than ST-DTDA.

Thus, in combining ST-DTDA with FMS and its extensions, as described above (described in Chapter 4), we meet all six of the requirements outlined in Section 1.3, to produce a scalable, fully decentralised task allocation solution mechanism which is robust to agent failure, adaptive to changes in the environment, efficient in communication and computation, provides good quality solutions and can provide bounds on the quality of the solutions it produces. Given this, in our final chapter, we conclude this thesis and outline some potential future work in this area.

Chapter 6

Conclusions and Future Work

In this thesis, we have developed a number of contributions to the field of multi-agent coordination for dynamic task allocation. In more detail, we first identified a need for better handling of scenarios in which the environment constantly changes, and methods to overcome computation and communication introduced by this. Given this, in Chapter 4, we presented fast-max-sum fast-max-sum (FMS), branch-and-bound fast-max-sum (BnB FMS) and bounded fast-max-sum (BFMS), three algorithms specifically tailored for use in task allocation environments. We have created these algorithms to use a particularly expressive factor graph formulation, in which factor nodes represent potential values for the variables to take, and so, modifying variable domains simultaneously simplifies the factor graph. For example, our formulation is such that an agent (a variable) can be assigned to (take the value of) a task, which is a factor node in the graph. It should be possible to formulate most other domains in this manner (for example, target tracking using sensors, radar scheduling, and pursuit evasion).

In more detail, FMS (introduced in Section 4.2) is an extension to the max-sum algorithm, which we showed to significantly reduce communication (up to 99% of the total message size sent) and computation (up to 99% of the total computation units used), when factor nodes are added to and removed from a factor graph. This, therefore, moves us further toward our requirements of adaptiveness (Requirement 4) and efficiency of communication (Requirement 3) from Section 1.3. However, these improvements come at a cost of storage at each agent which is linear in the number of agents and the number of tasks each agent can perform.

Next, BnB FMS (introduced in Section 4.3) further improves the efficiency of computation in FMS by removing values from variable domains which the variables will never take in an optimal solution. BnB FMS interleaves online domain pruning with FMS and applies branch-and-bound search when computing messages to be sent by a function in order to prune the state space to be explored further. Our empirical evaluation showed that solutions are found using up to 99% less computation and communication than

FMS (which achieves the same utility as BnB FMS), and require at least 25% fewer messages than branch-and-bound max-sum (BnB MS) in dynamic environments. However, this comes at the expense of a linear amount of storage at each agent, and lack of quality guarantees.

One key drawback of FMS is that we cannot guarantee the quality of the solutions it finds on factor graphs which contain cycles. To combat this, we presented BFMS in Section 4.4. BFMS is an efficient, anytime, superstabilizing extension of FMS which builds on ideas from bounded max-sum (BMS). In addition, BFMS is able to provide approximate solutions with quality guarantees (Requirement 6), whilst reducing redundant computation and communication in changing environments (Requirements 3 and 4). BFMS finds guaranteed solutions over cyclic graphs, unlike FMS. However, as with FMS, this comes at the cost of storage at each agent, and of the specific formulation mentioned earlier. In addition, in our empirical evaluation, we showed that BFMS introduces a tradeoff between communication and computation overheads, and solution quality. Finally, we evaluated the competitiveness of BFMS and FMS in terms of the number of variable values that are reassigned after changes in the environments, and found that BFMS provides an 80-100% improvement in this respect, by localising the impact of changes on the graph.

Whilst the expressiveness and simplicity of our formulation results in the communicational and computational savings gained by FMS and its extensions, it does lead to a problem of computation distribution — i.e., which agent should perform communication and computation for which task. In order for our algorithms to be truly decentralised, it was important to distribute their computation whilst avoiding creating computational bottlenecks. To solve this, we first identified that the scheduling on unrelated parallel machines problem (known as $R||C_{\max}$) from operations research was analogous to our own computation distribution problem. However, none of the existing algorithms to solve this problem are decentralised or adaptive to changing environments, and so, in Chapter 5, we presented spanning tree decentralised task distribution algorithm (ST-DTDA), which is the first decentralised algorithm for $R||C_{\max}$. ST-DTDA produces high quality approximations, with minimal run-time. In more detail, ST-DTDA uses heuristics to find an minimum/maximum spanning tree (MST) on which to run the min-max message-passing algorithm (which is from the generalised distributive law (GDL) family of algorithms), in order to produce good quality solutions efficiently, with a worst-case per-instance bound on solution quality. We showed how the key challenge for improving ST-DTDA is in choosing a heuristic which finds the right balance between solution quality and bound tightness.

In sum, ST-DTDA can be combined with FMS, BnB FMS and BFMS in order to provide a single, robust, standalone algorithm which can be applied to arbitrary dynamic task allocation problems. In this thesis, we have shown that this combination of approaches meets all six of the requirements laid out in Section 1.3 — a combination of

the approaches will be scalable, robust, efficient in communication, adaptive, and provide good quality approximate solutions with quality bounds. In terms of the broader field of multi-agent coordination, we have presented a complete suite of algorithms for use in real-world problems such as disaster management, target tracking using mobile sensors, and radar scheduling. Nevertheless, despite these accomplishments, we have identified a number of areas of future work that could further augment our a suite of dynamic task allocation algorithms for application in real world applications. We divide these into two groups: further theoretical work, and practical issues to be addressed for real world application. We discuss these in greater detail in the remainder of this section.

On the one hand, there are several items of theoretical work which we believe would build on the algorithms we have presented here. As such, we wish to address the following:

1. **Different utility functions** — In this thesis, we have focussed on problems where the global utility function can be factorised into the sum of local utility functions for each task. An interesting direction for future work would be to investigate other shapes of global utility functions and the impact that they have on our formulation in order to further generalise our algorithm. Solving this problem will require potentially re-formulating FMS itself or applying the techniques we introduced with FMS to another GDL algorithm.

2. **Fully connected graphs** — Graph sparsity is a key issue in this work, since the algorithms we use are particularly efficient on sparse graphs. While we have performed our empirical evaluation on graphs with varying densities, we have not addressed how to tailor our algorithms to perform better on complete graphs — i.e., environments where all agents can perform all tasks, such as disaster response on a small geographical scale — limited to a single building, for example. Thus, a final theoretical direction for work into our algorithms should address how our algorithms can be improved for connected graphs.

3. **Improved heuristics for ST-DTDA** — Earlier in this thesis, we presented three heuristics for ST-DTDA (see Chapter 5) and showed how they introduced a tradeoff between solution quality and bound tightness. Since this is the first decentralised algorithm for $R|C_{\max}$, it is unlikely that we have achieved the best possible result, and so there is space for further work into examining different heuristics and how they perform in different environments. This could produce heuristics which achieve both good solutions and tight bounds.

On the other hand, we must address a number of practical issues so that our algorithms can be deployed in real world applications such as the disaster management examples discussed throughout this thesis. These issues are:

1. **Degrees of predictability** — In this work, we have so far concentrated on completely unpredictable environments, where a completely myopic (i.e., only looking at the current state of the environment) view is the best the agents can feasibly manage. A new direction for this work would be to modify our algorithm at both function and variable nodes in order to consider environments with varying *degrees of predictability*: that is, environments where the agents could reason over and model how the environment could change in the next n timesteps with some varying degree of accuracy. This notion can be found in some real-world task allocation scenarios: for example, there are mathematical models representing the spread of fires throughout buildings, so this information could be used by firefighter agents to plan their actions based on where the fire is likely to spread to. Similarly, building collapse can be predicted to some extent if the composition of the building is known.

2. **Differing agent capabilities** — In this thesis, our model caters for agents having differing communicational and computational abilities, and differing abilities to perform tasks. Building on this, an interesting direction for this work would be to consider agents with differing capabilities or skills which must match the requirements for a given task. For example, in our disaster management example, some rescuers could have equipment to dig out buried civilians with, and some might not have this equipment. Thus, in this example, only rescuers with digging equipment should be sent to buried civilians. To do this we would need to modify agents and their variables such that variables can only take certain values depending on what their agent is capable of.

3. **Lossy communication** — Whilst we do address the problem of messages being lost before they are received, through timeouts and the re-sending of messages, there is space for further work. In more detail, future work could focus on the potential for messages to be changed for a number of reasons. For example, noisy communication channels, or malicious interference by a third party (particularly in environments following terrorist attacks, where communication channels could be intercepted and information changed). In addition to this, agents should be able to continue their decision making throughout prolonged periods of radio silence, by modelling the state of the environment and how it is likely to have evolved since the last communication.

By executing this future work, practical applicability of our algorithms can be further increased, which will move the state-of-the-art for dynamic task allocation another step closer to being applied in real-world scenarios.

Appendix A

GHS Pseudocode

In this appendix, we present the pseudocode for Gallager et al. (1983)'s GHS algorithm, for reference in our discussion of it in Section 2.3.4.

Algorithm A.1 Procedure `wakeup()`

```
1: let e_ij be adjacent edge of minimum weight
2: SE(e_ij) = Branch
3: LN = 0
4: SN = Found
5: find-count = 0
6: Send Connect(0) on edge e_ij
```

Algorithm A.2 Response to receipt of $Connect(lv)$ **message on edge** e_{ij}

```
1: if SN = Sleeping then
2:    execute procedure wakeup()
3: end if
4: if lv < LN then
5:    SE(e_ij) = Branch
6:    Send Initiate(LN, FN, SN) on edge e_ij
7:    if SN = Find then
8:       find-count = find-count + 1
9:    end if
10: else if SE(e_ij) = Basic then
11:    Place received message on end of queue
12: else
13:    Send Initiate(LN + 1, weight(j), Find) on edge e_ij
14: end if
```

Algorithm A.3 Response to receipt of $Initiate(lv, F, S)$ message on edge e_{ij}

```
1: LN = lv; FN = F; SN = S; in-branch = e_ij
2: best-edge = nil; best-wt = ∞
3: for e ≠ e_ij such that SE(e) = Branch do
4:    Send Initiate(L, F, S) on edge e
5:    if S = Find then
6:       find-count = find-count + 1
7:    end if
8: end for
9: if S = Find then
10:   execute procedure test()
11: end if
```

Algorithm A.4 Procedure `test()`

```
1: if there are adjacent edges in the state Basic then
2:    test-edge = the minimum weight adjacent edge in state Basic
3:    Send Test(LN, FN) on test-edge
4: else
5:    test-edge = nil
6:    execute procedure report()
7: end if
```

Algorithm A.5 Response to receipt of $Test(lv, F)$ message on edge e_{ij}

```
1: if SN = Sleeping then
2:    execute procedure wakeup()
3: end if
4: if lv > LN then
5:    Place received message on end of queue
6: else if F ≠ FN then
7:    Send Accept on edge e_ij
8: else
9:    if SE(e_ij) = Basic then
10:      SE(e_ij) = Rejected
11:   end if
12:   if test-edge ≠ e_ij then
13:      Send Reject on edge e_ij
14:   else
15:      execute procedure test()
16:   end if
17: end if
```

Algorithm A.6 Response to receipt of *Accept* message on edge e_{ij}

```
1: test-edge = nil
2: if weight(e_ij) < best-wt then
3:    best-edge = e_ij; best-wt = weight(e_ij)
4: end if
5: execute procedure report()
```

Algorithm A.7 Response to receipt of *Reject* **message on edge** e_{ij}

```
1: if SE(e_ij) = Basic then
2:     SE(e_ij) = Rejected
3: end if
4: execute procedure test()
```

Algorithm A.8 Procedure `report()`

```
1: if find-count = 0 and test-edge = nil then
2:     SN = Found
3:     Send Report(best-wt) on in-branch
4: end if
```

Algorithm A.9 Response to receipt of $Report(wt)$ **message on edge** e_{ij}

```
1: if e_ij ≠ in-branch then
2:     find-count = find-count − 1
3:     if wt < best-wt then
4:         best-wt = w; best-edge = e_ij
5:     end if
6:     execute procedure report()
7: else if SN = Find then
8:     Place received message on end of queue
9: else if wt > best-wt then
10:    execute procedure change-root()
11: else if wt = best-wt = ∞ then
12:    halt
13: end if
```

Algorithm A.10 Procedure `change-root()`

```
1: if SE(best-edge) = Branch then
2:     Send Change − root on best-edge
3: else
4:     Send Connect(LN) on best-edge
5:     SE(best-edge) = Branch
6: end if
```

Algorithm A.11 Response to receipt of $Change - root$ **message**

```
1: execute procedure change-root()
```

Appendix B

Flooding Scenario

In this appendix, we present the detailed scenario we use to inform our formal model given in Chapter 3. We base this scenario loosely on the 2007 flooding in Gloucestershire.

Having experienced one of the wettest Junes on record in Britain, the rain itself was not a shock, but the speed with which it was flooding the land around Chalford was. The rising waters had completely cut the village off by making transport into and out of it impossible, apart from by air. Running water had not been available for days, since the water treatment works had flooded, and the supply had long since been exhausted. The army had brought bottles and bowsers of drinking water in their helicopters, and even the beer companies had volunteered their tankers to transport water to fill the bowsers, but a form of black market had developed, with some members of the community taking more than their fair share and selling it on to others. The elderly and less able residents of the village had no choice but to purchase the water from this black market as they could not get to the bowsers fast enough before they were run dry.

Most residents of the village had moved into the church hall as their houses were no longer safe from flood water. They spent much of their time huddled round a battery-powered radio — their only source of information since the nearest electricity substation had to be turned off. Most people had not realised how much they relied on electricity-powered items to keep updated, from televisions, to the internet, to modern landline and mobile telephones whose batteries had run out with no power. Quite apart from the lack of electricity, the water had damaged a number of telephone lines around the village, cutting off communication, and residents had been told not to use mobile phones or walkie-talkies, so as to avoid overloading the networks and interfering with communication between emergency services.

On top of this, people were beginning to get ill. The standing water was encouraging rats, mosquitoes and flies and spreading disease. Many of the residents had been hit by diarrhoea, asthma, sore throats, cold sores, and chest problems, with the elderly being

the worst hit. There just wasn't enough clean water for people to take enough care with hygiene. RAF helicopters were being used elsewhere to rescue the sick and airlift them to hospital, but there simply weren't enough to rescue every single sick person. The limited available communication meant it was a case of waiting and hoping that rescue would come. Some people couldn't cope with the waiting — one such person tried using his petrol pump to pump water out of a cellar, filling the room he was in with carbon monoxide, causing his death by asphyxiation.

The situation for the emergency services wasn't much better than that in the villages. The RAF and Army were sparing as many helicopters and staff as they could to rescue people and reinforce riverbanks, but it wasn't enough. The vulnerable still needed transporting to places where they could be cared for, and land transport (such as ambulances) could not get to them. The lack of communication from the stricken areas meant the response agencies had no way of knowing where the most vulnerable civilians were. On top of this, the police were having to attempt to protect people's homes from opportunistic thieves exploiting the fact that they were left empty. The underlying problem throughout this effort, all of the agencies agreed, was the lack of collaboration between different agencies. Perhaps, if the Army and RAF could somehow communicate with the fire and ambulance service, qualified personnel could be airlifted into stricken communities, with medical equipment, in order to treat people's illnesses, rather than trying to airlift entire communities away. Collaboration between the fire and ambulance services would also prove useful in the swift rescue of civilians trapped in buildings rendered unsafe by the flood water — the fire service could work to stabilise the building, whilst medical personnel attend to those trapped inside.

The sort of technologies we are developing could be used to aid the response agencies in this scenario, to help support the decisions made by those agencies in dynamic situations. For example, if a helicopter arrived at an old peoples' home with the purpose of evacuating it, only to find that it had been evacuated beforehand, the helicopter pilot could use our technology to find the next best target without disrupting or informing too many other rescue vehicles. Another example would be if hospital staff called the emergency phone number to request that their hospital be evacuated as quickly as possible due to rising flood water. In this case, our technology could be used to find the best helicopters to attend the hospital without disturbing others that would not be able to get to it in time. A final, more morose, example would be if a helicopter got to an urgent building, only to find that all civilians in it had perished. Using our technology, the helicopter pilot could find out where to attend next without using valuable bandwidth communicating with other, far away, rescue agencies.

Bibliography

9/11 Commission. *The 9/11 Commission Report.* National Commission on Terrorist Attacks Upon the United States, 2004.

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

R. Albert and A. Barabàsi. Statistical mechanics of complex networks. *Review of Modern Physics*, 74(1):47–97, 2002.

J. Aspnes. Competitive analysis of distributed algorithms. In *Proceedings of the Thirty-Fifth Annual Symposium on Foundations of Computer Science*, pages 401–411, 1998.

Australasian Fire Authority Council. *The Australasian Inter-Service Incident Management System: A Management System for any Emergency.* 3rd edition, 2004.

R. Becker, S. Zilberstein, V. Lesser, and C. V. Goldman. Transition-independent decentralized markov decision processes. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–03)*, pages 41–48, New York, NY, USA, 2003. ACM.

M. Berhault, H. Huang, P. Keskinocak, S. Koenig, W. Elmaghraby, P. Griffin, and A. Kleywegt. Robot exploration with combinatorial auctions. In *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS–03)*, volume 2, pages 1957–1962, October 2003.

S. Bistarelli, R. Gennari, and F. Rossi. Constraint propagation for soft constraints: Generalization and termination conditions. In *CP*, pages 83–97, 2000.

C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge (TARK–96)*, pages 195–210, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

E. Campos-Nanez, A. Garcia, and C. Li. A game-theoretic approach to efficient power management in sensor networks. *Operations Research*, 56(3):552–561, 2008.

A. Chapman, R. A. Micillo, R. Kota, and N. R. Jennings. Decentralised dynamic task allocation: A practical game-theoretic approach. In *Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–09)*, pages 915–922, May 2009.

Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI–88)*, pages 49–54, 1988.

R. Dechter and D. Cohen. *Constraint processing.* Morgan Kaufmann, 2003.

E. W. Dijkstra. Cooperating sequential processes. *Programming Languages*, pages 43–112, 1968.

E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997(4):1–40, 1997.

A. Farinelli, A. Rogers, and N. Jennings. Bounded approximate decentralised coordination using the max-sum algorithm. In *Proceedings of the Tenth International Workshop on Distributed Constraint Reasoning (DCR–09)*, pages 46–59, July 2009.

E.C. Freuder and M.J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI–05)*, pages 1076–1078, 1985.

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

R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

C.E. Guestrin. *Planning under uncertainty in complex structured environments.* PhD thesis, Stanford University, 2003.

P. Gutierrez, P. Meseguer, and W. Yeoh. Generalizing ADOPT and BnB-ADOPT. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI–11)*, pages 554–559, 2011.

E Horowitz and S Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23:317–327, April 1976.

O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.

N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, M. J. Wooldridge, and C. Sierra. Automated negotiation: prospects, methods and challenges. *Group Decision and Negotiation*, 10(2):199–215, 2001.

F. V. Jensen. *An Introduction to Bayesian Networks*. Springer-Verlag, 1996.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

C. Kiekintveld, Z. Yin, A. Kumar, and M. Tambe. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–10)*, pages 133–140, Toronto, ON, Canada, 2010.

Y. Kim, M. Krainin, and V. Lesser. Application of max-sum algorithm to radar coordination and scheduling. In *Proceedings of the Twelfth International Workshop on Distributed Constraint Reasoning (DCR–10)*, pages 5–19, 2010.

R. D. Knabb, J. R. Rhome, and D. P. Brown. *Tropical Cyclone Report: Hurricane Katrina: 23–30 August 2005*. National Hurricane Center, 2006.

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

V. S. Anil Kumar, Madhav V. Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. A unified approach to scheduling on unrelated parallel machines. *Journal of the ACM*, 56:1–31, August 2009.

M.G. Lagoudakis, M. Berhault, S. Koenig, P. Keskinocak, and A.J. Kleywegt. Simple auctions with performance guarantees for multi-robot task allocation. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS–04)*, volume 1, pages 698–705, 2004.

J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

London Emergency Services Liason Panel. *Major Incident Procedure Manual*. TSO, 7th edition, 2007.

B. López, B. Innocenti, S. Aciar, and I. Cuevas. A Multi-agent System to Support Ambulance Coordination in Time-Critical Patient Treatment. In *Proceedings of the Seventh Simposio Argentino de Intelligencia Artificial (ASAI–05)*, pages 43 – 54, 2005.

B. López, B. Innocenti, and D. Busquets. A Multiagent System for Coordinating Ambulances for Emergency Medical Services. *IEEE Intelligent Systems*, 23(5):50–57, 2008.

D. J. C. Mackay. *Information theory, inference and learning algorithms.* Cambridge University Press, 2003.

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 *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–04)*, volume 1, pages 310–317. IEEE Computer Society, 2004.

R. McKie. Rescue mission for a nation under water. *The Observer*, page 2, 22 July 2007.

A. Meissner, T. Luckenbach, T. Risse, T. Kirste, and H. Kirchner. Design challenges for an integrated disaster management communication and information system. In *Proceedings of the First IEEE Workshop on Disaster Recovery Networks (DIREN–02)*, 2002.

Ministry of Civil Defence and Emergency Management. *The Guide to the National Civil Defence Emergency Management Plan.* 2007.

P. J. Modi, W. S. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence Journal*, 161 (1-2):149–180, 2005.

D. Monderer and L.S. Shapley. Potential games. *Games and economic behavior*, 14(1): 124–143, 1996.

R. Nair, M. Roth, and M. Yohoo. Communication for Improving Policy Computation in Distributed POMDPs. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–04)*, pages 1098–1105, Washington, DC, USA, 2004. IEEE Computer Society.

National Police Agency of Japan. *Japan: Damage Situation and Police Countermeasures associated with 2011 Tōhoku district - off the Pacific Ocean Earthquake, September 13, 2011.* 2011.

M. A. Palis. Competitive algorithms for fine-grain real-time scheduling. In *Proceedings of the Twenty-Fifth IEEE International Real-Time Systems Symposium (RTSS–04)*, pages 129–138, Washington, DC, USA, 2004. IEEE Computer Society.

C. Papadimitriou and J. N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operational Research*, 12(3):441–450, 1987.

T. Parsons. Pursuit-evasion in a graph. In *Theory and Applications of Graphs*, volume 642 of *Lecture Notes in Mathematics*, pages 426–441. Springer Berlin / Heidelberg, 1976.

P. Paruchuri, M. Tambe, F. Ordonez, and S. Kraus. Towards a formalization of teamwork with resource constraints. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–04)*, pages 596–603, Washington, DC, USA, 2004. IEEE Computer Society.

J. P. Pearce, R. T. Maheswaran, and M. Tambe. Solution sets for DCOPs and graphical games. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–06)*, pages 577–584, New York, NY, USA, 2006. ACM.

A. Petcu and B. Faltings. A distributed, complete method for multi-agent constraint optimization. In *Proceedings of the Fifth International Workshop on Distributed Constraint Reasoning (DCR–04)*, pages 1041–1048, 2004.

A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI–05)*, volume 19, pages 266–271. AAAI Press, 2005a.

A. Petcu and B. Faltings. S-DPOP: Superstabilizing, Fault-containing Multiagent Combinatorial Optimization. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI–05)*, pages 449–454, 2005b.

A. Petcu and B. Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI–07)*, pages 1452–1457, 2007.

M. Pitt. *Learning lessons from the 2007 Floods*. Pitt Review, 2008.

Provincial Emergency Program. *Introduction to Emergency Management in British Columbia*. 2007.

D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.

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

F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. ISBN 0444527265.

P. Scerri, A. Farinelli, S. Okamoto, and M. Tambe. Allocating tasks in extreme teams. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–05)*, pages 727–734. ACM, 2005.

O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101:165–200, May 1998.

R. Simon and S. Teperman. The world trade center attack: lessons for disaster management. *Critical Care*, 5(6):318, 2001.

D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

W. Smith and J. Dowell. A case study of co-ordinative decision-making in disaster management. *Ergonomics*, 43(8):1153–1166, 2000.

N. Stefanovitch, A. Farinelli, A. Rogers, and N. R. Jennings. Efficient multi-agent coordination using resource-aware junction trees (extended abstract). In *Proc. AAMAS–10*, pages 1413–1414, 2010.

R. Stranders, A. Farinelli, A. Rogers, and N. Jennings. Decentralised coordination of mobile sensors using the max-sum algorithm. In *Proc. IJCAI–09*, pages 299–304, 2009.

US Department of Homeland Security. *National Incident Management System*. 2004.

W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961.

M. Vinyals, M. Pujol, J. A. Rodríguez-Aguilar, and J. Cerquides. Divide and Coordinate: solving DCOPs by agreement. In *Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS–10)*, pages 149–156, Toronto, ON, Canada, 2010.

J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton, 1944.

M. Wainwright, T. Jaakkola, and A. Willsky. Tree consistency and bounds on the performance of the max-product algorithm and its generalizations. *Statistics and Computing*, 14(2):143–166, 2004.

A. Waldock, D. Nicholson, and A. Rogers. Cooperative control using the max-sum algorithm. In *Proceedings of the Second International Workshop on Agent Technology for Sensor Networks (ASTN–08)*, pages 65–70, 2008.

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

A. Wotzlaw. *Scheduling unrelated parallel machines: algorithms, complexity, and performance*. PhD thesis, Universität Paderborn, 2006.

W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

Z. Yin, C. Kiekintveld, A. Kumar, and M. Tambe. Local Optimal Solutions for DCOP: New Criteria, Bound and Algorithm. In *Proceedings of the Second International Workshop on Optimisation in Multi-Agent Systems (OPTMAS–09)*, Budapest, Hungary, 2009.

M. Yokoo, T. Ishida, E. H. Durfee, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems (ICDCS–92)*, pages 614–621, 1992.

X. Zheng and S. Koenig. Reaction functions for task allocation to cooperative agents. In *Proc. of AAMAS 2008*, pages 559–566, 2008.