
School of Technology and Architecture
Department of Information Science and Technology

# Distributed Coordination in Unstructured Intelligent Agent Societies

António Luís Morais Costa da Silva Lopes

A Dissertation presented in partial fulfilment of the Requirements for the Degree of
*Doctor of Information Science and Technology*

**Advisor**:
PhD, Luís Miguel Botelho, Associate Professor
*Lisbon University Institute*
*(Department of Information Science and Technology)*

September 2010

# Abstract

Current research on multi-agent coordination and distributed problem solving is still not robust or scalable enough to build large real-world collaborative agent societies because it relies on either centralised components with full knowledge of the domain or pre-defined social structures. Our approach allows overcoming these limitations by using a generic coordination framework for distributed problem solving on totally unstructured environments that enables each agent to decompose problems into sub-problems, identify those which it can solve and search for other agents to delegate the sub-problems for which it does not have the necessary knowledge or resources. Regarding the problem decomposition process, we have developed two distributed versions of the *Graphplan* planning algorithm. To allow an agent to discover other agents with the necessary skills for dealing with unsolved sub-problems, we have created two peer-to-peer search algorithms that build and maintain a semantic overlay network that connects agents relying on dependency relationships, which improves future searches. Our approach was evaluated using two different scenarios, which allowed us to conclude that it is efficient, scalable and robust, allowing the coordinated distributed solving of complex problems in unstructured environments without the unacceptable assumptions of alternative approaches developed thus far.

**Keywords**: Multi-agent systems, distributed planning, coordination.

# Resumo

As abordagens actuais de coordenação multi-agente e resolução distribuída de problemas não são suficientemente robustas ou escaláveis para criar sociedades de agentes colaborativos uma vez que assentam ou em componentes centralizados com total conhecimento do domínio ou em estruturas sociais pré-definidas. A nossa abordagem permite superar estas limitações através da utilização de um algoritmo genérico de coordenação de resolução distribuída de problemas em ambientes totalmente não estruturados, o qual permite a cada agente decompor problemas em sub-problemas, identificar aqueles que consegue resolver e procurar outros agentes a quem delegar os sub-problemas para os quais não tem conhecimento suficiente. Para a decomposição de problemas, criámos duas versões distribuídas do algoritmo de planeamento Graphplan. Para procurar os agentes com as capacidades necessárias à resolução das partes não resolvidas do problema, criámos dois algoritmos de procura que constroem e mantêm uma camada de rede semântica que relaciona agentes dependentes com o fim de facilitar as procuras. A nossa abordagem foi avaliada em dois cenários diferentes, o que nos permitiu concluir que é uma abordagem eficiente, escalável e robusta, possibilitando a resolução distribuída e coordenada de problemas complexos em ambientes não estruturados sem os pressupostos inaceitáveis em que assentava o trabalho feito até agora.

**Palavras-chave**: Sistemas multi-agente, planeamento distribuído, coordenação.

To Vânia and Francisco.

# Acknowledgements

For most people, a PhD is a lonely, stressful (and sometimes painful) hard-working phase of their lives. I have to say that, for me, this was not the case. And the main reason is because I was lucky enough to have a lot of people that I could count on to help make this a very smooth and enjoyable ride.

And if there is one person to whom I am thankful for this is my advisor, Professor Luís Botelho. It all began, in January 2000, when he decided to contact some of the most promising students of his Artificial Intelligence course at ISCTE to come work at the ADETTI research lab. I was *not* one of them. But since I was in the classroom when he did contact those students, I decided to propose myself to take the tests anyway and I was lucky enough to be amongst the two students that were chosen. We have been working together ever since. And, oddly enough, I have my eavesdropping and rudeness to thank for because, had I not decided to barge in the conversation and propose myself to take the tests, I would not have had the privilege of having such a wonderful and rewarding professional life alongside Professor Botelho. This is an important milestone of the work we have done in the last few years while working on research projects, on the M.Sc. thesis, of our discussions in our weekly meetings and occasional project meetings in some pleasant European city and sometimes over some very inspirational drinking sessions. It would be selfish to say the work is only mine. That is why you, the reader, will see the words "we", "us" and "our" throughout the entire thesis. Because the work described in this thesis is not my work, it is *our* work. Thank you

so much for your guidance, wisdom and friendship, Luís. I could not have done it without you.

I wish to thank my fellow colleagues at ADETTI and ISCTE-IUL: Pedro Figueiredo, Luís Nunes, Filipe Santos, Ricardo Ribeiro, Rui Lopes, Sancho Oliveira, Manuel Sequeira, Anders Christensen, Alexandre Almeida, José Farinha, André Santos, Jorge Louçã, Bruno Gonçalves and Sofia Esteves. They were patient enough to hear my ideas, discuss them and deliver truly constructive criticism that only enriched the research. Special thanks go to Luís Nunes and Filipe Santos because they managed to make my life as a teaching assistant a lot easier by taking up some of the work that I should have done in the courses that they coordinate.

I wish to thank Fátima Silva, Sónia Ferro, Ana Rita Leitão, Fátima Estevens and Gabriela Almeida for making my life so much easier when it comes to having to deal with the bureaucratic elements of academia and research.

I wish to thank Professor Steven Willmott (from Universitat Politécnica de Catalunya, in Barcelona, Spain), Professor Sascha Ossowski (from Universidad Rey Juan Carlos, in Madrid, Spain) and Professor Edmund Durfee (from the University of Michigan, in the United States of America - mentor at the Doctoral Mentoring Program at the AAMAS'07 conference) for providing useful knowledge at the beginning of this PhD that helped me not wander around trying to solve all the problems in the world and focus on a particular subject. Special thanks go to Professors Pedro Ramos (from ISCTE-IUL), Willmott and Ossowski for writing recommendation letters for my application to the FCT PhD scholarship.

The work performed on this PhD was greatly motivated by the work done in the European projects Agentcities and CASCOM. I would like to thank our former project partners for supporting our work and help discussing alternative approaches: Thorsten Moeller,

Alberto Fernandez, César Cáceres, Oliver Keller, Heikki Helin, Bastian Blankenburg, Michael Schumacher, Heimo Laamanen, Ahti Syreeni, Matteo Vasirani, Matthias Klusch, Alexandre de Oliveira e Sousa, Federico Bergenti, Heiko Schuldt, Nadine Froehlich, David Bonnefoy and Ion Constantinescu.

I wish to thank Professor Rui Lopes (from ISCTE-IUL) and Professor Benedita Malheiro (from the Politechnical Institute of Porto) for providing valuable feedback on a preliminary and shortened version of this thesis that helped shape it into a full dissertation. I also would like to thank them and Professor Helder Coelho for accepting to be a part of the jury of this PhD.

This work has been supported in part by the Portuguese Foundation for Science and Technology (Fundação para a Ciência e Tecnologia) under the scholarship grant SFRH/BD/27533/2006. I would also like to thank the support of the Lisbon University Institute (ISCTE-IUL), ADETTI (Associação para o Desenvolvimento das Tecnologias e Técnicas de Informática) and the Instituto de Telecomunicações.

This life project of embarking on a PhD would not be possible if it wasn't for the support of the most important people in my life, my family and friends. They are the foundation from which I am able to build such a rewarding professional and personal life. I deliver my eternal gratitude:

> To all my friends that gave all the necessary moral support and were wonderfully enthusiastic even when they had to face my boring explanations.
>
> To my brother Rui and sister-in-law Leonor because, not only we had such a wonderful time in all of our occasional getaways, lunches, dinners, trips and assorted social events, it was great to have someone that was also going through the same to share the experience with (and learn from each others' mistakes).

To my parents, António and Fátima, for their unconditional love and support and for constantly showing how proud they are of me.

To my wife, Vânia, for her patience and understanding towards my work (which included cheering me up when I wondered around with a grumpy face) and for completing my life in such a beautiful, caring and loving way.

And finally to my son, Francisco, for showing me how life can be so fulfilling with just a little smile.

# Contents

# List of Figures

# List of Tables

# Listings

# Glossary

***Graphplan*** The first automated planner that was based on the concept of a planning graph. 18, 19, 68, 73, 75, 83–85, 92–94, 112, 113, 115, 117, 121, 122, 124, 125, 131

***mutex*** *Mutually exclusive* – Expression used to refer to the fact that two actions or propositions cannot occur at the same time in a planning environment. 18, 72–74, 76–78, 82, 84, 94

`no-op` *No-operator* – An expression used to refer to the absence of an operator in a planning graph. 17, 71–73, 82, 84, 95

**AI** *Artificial Intelligence*. 7, 12

**CASCOM** *Context-aware Business Application Service Co-ordination in Mobile Computing Environments* – An European Commission funded project in which we participated. 2–4

**CPU** *Central Processing Unit*. 109

**DFS** *Depth First Search*. 62, 90

**DHT** *Distributed Hash Tables*. 32–34

**FIFO** *First In, First Out*. 96, 97

**HTN** *Hierarchical Task Networks*. 21

**IBDFS** *Iterative Branching Depth First Search*. 59, 62, 90, 101, 103, 104, 106, 107, 109–112, 121, 122, 124

**LIFO** *Last In, First Out*. 96, 97

**MAS** *Multi Agent Systems*. 32, 34, 35

**NN** *Number of Neighbours*. 101, 108

**P2P** *Peer-to-peer*. 6, 9, 11, 12, 29–37, 40, 44, 46, 50, 64, 77, 123, 124

**PbF** *Priority-based Flooding*. 59, 60, 90, 101, 121, 124

**PDDL** *Planning Domain Description Language*. 41, 42, 69, 70

**POP** *Partial Order Planning*. 92, 94

**RDF** *Resource Distribution Factor*. 105–108

**STRIPS** *STanford Research Institute Problem Solver*. 13, 71

**TOBC** *Totally-Ordered Backward-Chaining* planner. 92, 93

**TOFC** *Totally-Ordered Forward-Chaining* planner. 92, 94

**TTL** *Time-To-Live*. 30, 77, 101–104, 106, 108, 109, 111

# Chapter 1

# Introduction and Motivation

Most real-world problems are too complex to be solved by individuals working in isolation, due to the lack of necessary expertise, resources or information. One of the powerful motivations for distributed problem solving is that it is difficult to build an agent to be competent in every possible task. Moreover, even if it is feasible to build (or train) an omni-capable agent, it is often overkill because, at any given time, most of those capabilities will go to waste (Durfee, 1999). Combining different expertise to solve problems that are beyond each individuals capabilities may be the right strategy to solve complex problems. However, without coordination, distributed problem solving may become impossible.

The research work described in this thesis strives to answer (affirmatively) to the following: is it possible to create a robust, efficient and scalable system to coordinate the distributed problem solving activity of multiple heterogeneous agents in unstructured environments? The relevance of our question and, therefore, of our work comes from our research experience of the last 8 years, in which we have developed several kinds of approaches to allow agents to seamlessly cooperate with each other in complex coordination activities. Our previous work and the work of others were based on structured (centralised, hierarchical, social or organisational) systems that have robustness and scalability issues.

The limitations of structured environments motivated us to focus our work on totally unstructured distributed environments and create a more flexible, generic and robust approach to contribute to the deployment of real-world intelligent agent societies. Our goal is to develop a robust, scalable and efficient distributed

coordination framework that allows agents to decompose problems into sub-problems, identify and solve the sub-problems for which they have the necessary skills, discover agents with the potential to solve the remaining sub-problems and send them the partially-solved problem. Each agent uses this process until the problem is completely solved.

The first two sections of this introductory chapter summarise the work that we (and our project partners) have done in two different projects funded by the European Commission, the Agentcities project (described in section 1.1) and the CASCOM project (described in section 1.2). These two projects represent what has been the mainstream approach to solving the coordination problem in large-scale distributed environments, *e.g.*, using structured or centralised elements. Section 1.3 summarises the limitations of these approaches by pointing out potential robustness and scalability issues.

We then briefly present, in section 1.4, our alternative approach that proposes the combination of peer-to-peer computing and distributed problem solving to allow agents to collaborate in solving complex problems without the need for structured elements of any kind. In particular, we propose the use of artificial intelligence planning algorithms to allow agents to determine which parts of a problem they can contribute to and the use of efficient peer-to-peer search algorithms to find agents that can contribute to solving the remaining parts. Finally, section 1.5 presents the structure of the remainder of this thesis.

## 1.1 The *Agentcities* Project

Coordination, communication, discovery, trust, security and ontology issues are some of the challenging elements that are found in truly open environments where agents, owned by many different individuals and organisations, can interact and interoperate. The *Agentcities* Project (Willmott *et al.*, 2001) was an initiative to create a global, open, heterogeneous network of agent platforms and services to which any agent researcher could connect his or her agents. The actual network was built through the deployment of several agent platforms throughout the world, based on the cities where the project partners were operating. Each platform contained mandatory management agents and a set of application specific

agents that provided services in several domains ranging from travel, tourism and entertainment services to marketplaces and payment systems.

While the main focus of the project was to address interoperability and openness issues, the domains of the services available in the network were so diverse that the environment was an obvious candidate to explore the large-scale coordination of multi-agent-based services. One of the attempts of providing a composition service (combining several different services into one value-added compound service) was based on a template mechanism (Dale & Ceccaroni, 2002). These template-based planning processes were very specific and aimed at providing fast and well-defined services based on user preferences. However, this required that new templates had to be manually created for each new domain or integrated service that the system would provide.

This raised the need to create a dynamic composition process that would be as independent as possible of the particular application domain in which it would operate. The solution was to develop an ontology-based broker agent (Botelho *et al.*, 2003) capable of searching information from various sources, pertaining diverse topics, integrating it in coherent ways and evaluating it (using a fuzzy logic mechanism) according to specified user preferences. Even though this approach was generic enough to be used in several different domains, it still had some major drawbacks. The composition process was totally centralised, which not only originated a central point of failure but also accumulated a lot of work (like gathering many different pieces of information) onto a single entity. Although it was possible to have several broker agents of this type, the replication was futile since the agents could not cooperate in the information gathering process. Another disadvantage resided on the fact that it could only be used for information services using well-defined ontologies. Moreover, the similarity of concepts between ontologies could lead to using information sources that were completely irrelevant to the problem at hand.

## 1.2 The CASCOM Project

The main goal of the CASCOM project (Helin *et al.*, 2005) was to create a supportive infrastructure for business application services in which these were flexibly

coordinated and pervasively provided to the mobile user by intelligent agents in dynamically changing contexts of open and large-scale environments. One step towards this vision was the development of an agent-based service coordination system. This system included several domain-independent coordination agents responsible for the discovery (Schumacher *et al.*, 2008), matchmaking (Botelho *et al.*, 2008a), composition (Blankenburg *et al.*, 2008) and execution (Botelho *et al.*, 2008b) of semantic web services.

Service composition, being one of the most complex problems within service coordination, was addressed in the project by using an artificial intelligence (AI) planner to create the desired compound service. AI planners require the set of all available actions (also referred to as action or planning operators) that can be used to create the compound service. In the realm of service composition, planning operators (i.e., available actions) are the services that may be considered to be included in the final value-added compound service. This means that, using the CASCOM's approach, it would be necessary to provide the set of all relevant available services to the central planner that ultimately would create the compound service.

Although conceptually simple, this turns out to be a very complex problem. The main challenge being to locate only the relevant services, which is an impossible problem to solve, in the general case. One solution would be to provide the planner with the set of all existing service descriptions. This is not a feasible option because the set of all existing services, in a large-scale environment, tends to be gigantic, rendering the approach impracticable. A different option would be to restrict the existing services to those of the categories (or using the ontologies) related to the service to be composed. This option is also problematic since, as concluded in the previous section, it is often impossible to determine the set of all relevant categories to the creation of the compound service. Some compound services may include services from various unrelated categories, making it impossible to accurately determine which categories are to be included in the process.

The CASCOM Project tried also a different approach, namely the use of context aware computing (Costa *et al.*, 2008). In this approach, context information would be used to restrict the set of services to be considered by the AI planner.

Unfortunately, it was also impossible to devise a general approach that could effectively reduce the number or returned services without risking gathering sets of services that would be insufficient to create the desired compound service, since it is not yet known how context information should be used in general purpose agents. Similarly to what happened in the *Agentcities* project, this approach presented some drawbacks mainly because it relied on single centralised entities to perform main coordination tasks. Each service coordination agent would have to know all existing services (ideally, only those relevant to their operation) and their descriptions, which is not an option.

## 1.3 Limitations of Structured Systems

The research on intelligent agents has devoted considerable effort not only to communication and coordination, but also to reasoning, learning, and adaptation of each agent, seeking to increase their autonomy. However, multi agent systems often suffer from incapability to coordinate themselves in dynamic environments where no structure is present (Küngas & Matskin, 2006). The two projects described above constitute good examples of what has been the mainstream research in the field of distributed problem solving for the past few years. The use of structured systems has been the obvious choice mainly because they are simpler and faster to implement. However, the dependency on centralised or structured components, while it simplifies the deployment of coordination environments, has some major drawbacks.

On one hand, the main components of such systems can turn into *bottlenecks* and potentially catastrophic points of failure. Unless they are very resourceful (large processing power – which, in turn, has high maintenance costs) and robust (able to withstand failures) they will not be able to keep up with the growth of the environment in which they operate. On the other hand, structured systems usually require previous knowledge (often provided by the human user) to be deployed. This refers not only to the main components, but also to the individual agents that, wanting to connect to such systems, must have prior knowledge of the organisation of the main components in the system, its protocols, languages and used ontologies.

Our approach pursues the alternative of using unstructured networks to deploy fully distributed cooperative environments. However, these have some disadvantages of their own. For example, agents operating in these environments, not being able to rely on structured elements, need efficient search mechanisms to find the resources they require to solve specific problems. This is one of the areas in which peer-to-peer (P2P) research has been focusing on the past few years and one that we believe can help develop efficient and scalable distributed problem solving systems on top of unstructured networks.

## 1.4 Peer-to-Peer Computing and Distributed Problem Solving

The evolution of peer-to-peer (P2P) research has reached promising results, paving the way for developing more robust and scalable applications on top of P2P networks. Nonetheless, even though P2P computing presents some interesting properties that would enable creating high performance applications, it still lacks the degree of proactivity that would enable higher autonomy, and rationality (Küngas & Matskin, 2006). Research on P2P computing has mainly addressed the efficient management of the network, treating each peer as a simple reactive node, with little or no autonomy at all, thus ignoring the potential for developing collaborative environments.

The combination of the distributed capabilities of P2P networks with the intelligence of autonomous agents seems promising since it will allow the transparent access to large-scale distributed resources while maintaining high availability, fault tolerance and low maintenance application deployment through self organisation (Willmott *et al.*, 2005). We use both technologies to create an intelligent P2P infrastructure that will enable a dynamic network of intelligent agents to cooperatively and efficiently discover and coordinate several resources to solve faced problems.

Our main goal is to develop a robust, efficient and scalable agent architecture that enables agents to freely participate in distributed problem solving in unstructured societies. In such societies, agents receiving requests from other

agents would be capable of using their own capabilities to handle the part of the problem for which they have competence and resources, and distribute the partially solved problem to other agents that can possibly provide further contributions. This work combines multi-agent systems and peer-to-peer computing in order to address its two main challenges:

- In very large networks it is impracticable for one agent to know which agent skills exist at a given moment. Therefore, each agent has to be equipped with an AI-based algorithm capable of planning with only partial knowledge of available skills, that is, they will be able to contribute to solving only a part of the problem, for which they have know-how, but the remaining parts will remain unsolved and thus will be sent to other agents that may contribute to them. Current state of the art planning algorithms, while very efficient, are dependent on having full knowledge of a domain's planning operators to solve specific planning problems. Otherwise, they fail without providing any information as to which part of the problem they are able to solve.

- Since, in general, agents will only be able to contribute to parts of a specific problem, they have to be able to discover the most suitable agents to which the partially solved problem should be forwarded. However, current state-of-the-art network search algorithms are unable to efficiently and robustly discover specific agents in large unstructured networks since they make use of central entities, hierarchical structures or social rules to address the inefficiencies of unstructured algorithms. In doing so, these algorithms become prone to failures that can compromise an entire (or at least a very significant part of the) system.

## 1.5 Thesis Overview

In this section, we present an overview of the thesis with the goal of providing an overall summary of the research work to the reader. The introduction sets the research goal of the thesis and the tone to the work that is described in the remainder of the document. Our goal is to test and evaluate the hypothesis of

deploying a coordination environment for agent-based distributed problem solving that relies on a totally unstructured network. This goal was motivated by the shortcomings of current research in this field that mainly relied on structured systems. To fully expose these limitations, following the introductory chapter, we present in Chapter 2 a thorough review of state-of-the-art research in the field by focusing on the limitations and challenges of related work and by identifying the aspects on which our research work contributes to advance the state of the art.

The conclusions of Chapter 2 are two-folded. On one hand, we establish that classical planning (in section 2.1) and current research on distributed coordination and planning (in section 2.2) are not adequate to efficiently and robustly deal with very large distributed environments, mainly due to the use of centralised components and pre-defined organisational or social structures. On the other hand, distributed problem solving in a large unstructured network requires efficient peer-to-peer search mechanisms and dynamic self-organisation methods, which current research on distributed artificial intelligence still fails to deliver (as stated in sections 2.3 and 2.4). The chapter ends by summarising the limitations and challenges of the analysed approaches and outlines the main research contributions of this thesis (in section 2.5).

In Chapter 3, we describe and analyse two different testing scenarios, both of which demonstrate distributed problem solving approaches in environments larger than those used in concurrent approaches. Each scenario explores different aspects of the overall distributed problem solving process. On one hand, the *Rescue Agents* scenario (described in section 3.1) is a very complex and demanding environment from the cooperation point of view. Most problems in this scenario require agents to often interact with each other, that is, the level of interaction and dependency between the agents is very high. On the other hand, the *Custom Balls Factory* scenario (described in section 3.2) is less complex from that cooperation point of view but it is highly demanding from the discovery point of the view. Most problems in this scenario are easily solved with the participation of only a few agents. The challenge is to find the appropriate agents from a very large list of potential candidates. The goal of this chapter is, not only, to show how these scenarios' problems could be addressed in a distributed problem

solving environment, but also to accurately determine the requirements of such environments. Hence, Chapter 3 ends with a detailed analysis (in section 3.3) of the requirements for a technical approach suitable to deal with problems from these scenarios.

Driven by the conclusions of the review of related work (in Chapter 2) and the analysis of the testing scenarios (in Chapter 3), the main requirement of our research is the development of a framework for robust and scalable collaborative distributed problem solving and coordination in large, unstructured agent networks. This framework, which is described in Chapter 4, includes two major components. One of the components (described in section 4.1) is a set of P2P search algorithms that discover the agents to which the unsolved sub problems will be delegated. These P2P algorithms dynamically build and update a self-organised semantic overlay network that greatly speeds up the discovery process. The other component of the framework is a distributed partial planning algorithm (described in section 4.2), which allows an agent to identify and solve parts of a problem for which it has enough skills and resources, and to identify those other parts that need to be delegated to other agents.

The framework described in Chapter 4 does not resort to pre-imposed organisational structures, social laws, or centralised components. This means that agents equipped with the distributed planning and the P2P algorithms of the proposed framework can integrate very large and totally unstructured agent societies and collaboratively and efficiently solve complex problems.

Chapter 5 shows that the proposed framework is indeed robust, scalable and efficient. This chapter presents the tests we have performed and the results obtained in the two different testing scenarios described in Chapter 3. We conclude that the system is robust, since the forced removal of some nodes throughout the execution of some tests showed little negative impact on the performance of the distributed system. Regarding scalability, this chapter also shows that the system's performance, using the appropriate planner, is proportional to the size of the problem to be solved. Moreover, workload tests show that our approach uses fewer resources to achieve faster results, thus allowing us to conclude that it is in fact an efficient system.

These are the major achievements of our research work, which are also outlined in Chapter 6 (in section 6.1), the concluding chapter of this document. We also present (in section 6.2) some of the limitations of the chosen approach and discuss how these can be addressed in future work. Namely, the possibility of exploring alternative (and potentially better) solutions by forcing agents to search for extra participants that can solve parts of the problem in parallel. Also, we discuss how our approach can perform faster if we assume agents can act on behalf of other agents if they hold the necessary knowledge about them. Moreover, we speculate how a context-aware approach could be used to improve the selection of the agents to which the partially-solved problem should be forwarded.

A list of the published papers during the course of our research work is presented in Appendix A.

# Chapter 2

# Related Work

One of the major challenges of creating real-world agent societies is to develop a coordination infrastructure that is scalable and robust enough to support increasingly complex problems. Multi-agent coordination has been the focus of much research in the area of distributed problem solving and multi agent systems. Although extensive research has been done in this area, there is still room left for improvement. In the introductory chapter, we have proposed the combination of multi-agent coordination and peer-to-peer (P2P) computing as an efficient way of deploying scalable and robust distributed problem solving.

This chapter addresses the analysis of related research in the areas pointed above. P2P computing is thoroughly described in section 2.3, while multi agent based coordination in distributed environments is reviewed in detail in section 2.2. However, since multi-agent based coordination involves splitting problems into several pieces and distributing them to agents with appropriate skills, thus relying on some kind of planning algorithm, we present a brief tutorial of planning approaches for the single agent case. Hence, in section 2.1, we review the fundamentals of automated planning by describing the major algorithms and their limitations. These algorithms were designed for being used by a single agent with full and deterministic knowledge of its domain and environment. However, some of them can be adapted to be used in distributed environments.

In section 2.2, we analyse current coordination and planning approaches designed for multiple agents collaborating to solve complex problems. This analysis

mainly concludes that, even though these approaches are able to address the general problem of coordination for distributed planning, they do so by making use of centralised components or organisational-based structures that, in the event of sudden failure, can compromise the entire system.

Considering the potential of P2P computing to address the efficient management of large networks, we review, in section 2.3, the fundamental aspects of network search algorithms and techniques in order to determine how these can be used to help multi-agent coordination approaches avoid centralised elements or superimposed organisational structures and deploy totally decentralised and efficient societies of agents. Although P2P computing does hold this potential, this section points out that some of the algorithms and techniques can be further improved, namely, by introducing semantic information.

In section 2.4, we analyse hybrid approaches that combine semantic based and multi-agent coordination with P2P computing. This section mainly shows that such approaches do not fully explore the potential of semantic information or do so very inefficiently.

Finally, in section 2.5, we summarise the analysis by discussing the limitations of current research and by presenting the challenges that define our research contributions, inline with the goals set in the introductory chapter.

## 2.1 Automated Planning

For several years, *Artificial Intelligence* (AI) has been trying to create intelligent autonomous entities that perceive their environment and act upon it to achieve their goals. With the aim of providing agents with the necessary capabilities to define strategies or action sequences that lead them closer to their objectives, Automated Planning became one of the main research fields of AI.

A typical planner uses an algorithm that takes the initial state of the world, a description of one or more goal states and a set of available actions (also called operators) and generates an ordered sequence of actions capable of leading the agent from the initial state to a goal state. There are essentially three types of planning algorithms, according to the search space in which they operate:

state-space planning, plan-space planning and planning-graph planning. These algorithms are further described in the following sub-sections.

### 2.1.1 State-space Planning

Fikes & Nilsson (1971) have been the first to create a formal description of a planner, called STRIPS (STanford Research Institute Problem Solver), with the characteristics described above. STRIPS is an automated planner whose name was later used to refer to the formal language used to describe the inputs for a classical planning problem. The formal specification of STRIPS allows planning problems to be represented as state-space graphs, where nodes represent the different possible states of the domain (each state is represented by a set of propositions that hold true at that state) and arcs are state transitions (or actions). In such a representation, a plan is a sequence of actions corresponding to a path (in the state-space graph) from the initial state to a goal state.

The state-space graph provides a useful abstraction for finding plans. However, it is important to define a strategy that allows finding the best plan faster. There are two main classes of algorithms that search plans in the described state-space: *forward-chaining* and *backward-chaining* algorithms.

The algorithms differ basically in the direction in which they search the state space graph. *Forward-chaining* finds actions that can be chained together starting from the initial state leading to the goal state, whereas *backward-chaining* applies the reverse process, starting from the goal state and finding actions that can be chained together until the initial state is reached. The *forward-chaining* algorithm determines (starting from the initial state) if, in each iteration, the goal state has been reached. If so, then the current plan is returned. Otherwise, it chooses an action (from the set of available actions that can be applied to the current state, that is, actions whose pre-conditions are satisfied by the conditions of the current state) to produce a partial solution for the next iteration. The *backward-chaining* algorithm starts at the goal state and inversely applies available actions to produce sub goals (the preconditions of the applied actions), stopping if it produces a set of sub goals that are satisfied by the initial state.

In each iteration of the algorithms, an action is chosen to be applied to the current state. Unfortunately, choosing the appropriate action that will lead to the

shortest (or the less expensive) plan is not a trivial task. Some of the techniques that can be employed are the *breadth-first*, *depth-first*, *best-first* and *branch-and-bound* searches. Even though all of these techniques can be applied to both algorithms, for the sake of simplicity, we will only show how they apply to *forward-chaining*.

A *breadth-first* search starts by generating nodes that represent the states produced by the application of all actions to the initial state and by checking if any reaches the desired goal. Nodes to which all actions have been applied are termed expanded nodes. If the goal is not reached, the search is continued in the state produced by the action that was used first. Again, if the generated nodes do not represent goal states, the search is carried out, from the yet to be expanded node that was generated earlier.

The distinctive feature of the *breadth-first* search is that the next node to be expanded is always selected from the shallowest nodes not yet expanded, which ensures that, if it exists, the shortest plan is always generated. The process is carried out until the goal state is found at some level of the state-space graph. Although the shortest plan is guaranteed to be found if it exists, this algorithm has huge memory requirements. If there are `n` potential actions and the goal nodes are found at depth `d`, the required memory space will be $n^d$ times the average amount of memory required for each state. Since the time taken by the algorithm to generate the plan is proportional to the generated number of nodes, it is also proportional to $n^d$.

A *depth-first* search does exactly the opposite of the *breadth-first* search by giving priority to exploring deepest nodes prior to exploring shallowest nodes. It applies one specific action to the initial state and if the desired goal state has not been achieved yet, then it chooses one action to apply at the state produced in the first step (the second depth level). This search process continues until either a solution is found or no more actions can be applied to the state at the current level, in which case it backtracks to the previous level with unvisited nodes.

In the worst case, a *depth-first* search may need to examine the entire search space before finding a solution. Therefore, its running time may be worse than that of a *breadth-first* search (Ghallab *et al.*, 2004). However, because a *depth-first* search keeps track of only the nodes on the current path, its space/memory

requirement is only equal to the depth of the deepest node it visits. Unfortunately, the *depth-first* search is not guaranteed to find a solution, even if one exists, let alone an optimal solution.

If some knowledge of the domain exists and can be used to help choose the appropriate nodes to be visited next, then these search techniques can be greatly improved. *Best-first* search uses a heuristic approach based on a deterministic function that helps decide which node on the state-space graph should be visited next. The *best-first* search also maintains a list of nodes that have been generated but not yet visited. However, instead of using this list as a queue the way other search techniques do, it uses the list as a priority queue: the next node chosen from the list will be the one with the best value of the heuristic function. The best value may be the smallest (in case of a *minimisation* function) or the largest (in case of a *maximisation* function) depending on the domain dependent heuristic function that is being used.

The *branch-and-bound* search (also referred to as $A*$) is a particular case of a *best-first* search technique. It uses a heuristic function that considers the cost of the built plan so far, plus an estimative of the cost of the rest of the plan (to reach the goal state) to decide which node should be considered next. Additionally, it uses that function to eliminate nodes that it does not need to visit (also called *pruning*). In this technique, a global variable that holds the best solution seen so far is kept. If the best estimate of the current node is already worse than the best solution seen so far, then there is no need to visit its child nodes. This node is pruned and, thus, the search space is reduced.

### 2.1.2 Plan-space planning

State-space planning views a plan as a mere sequence of actions that achieves a desired goal state, given an initial state and a set of potential actions to be used. In most real world problems this view is too simplistic and it is not suitable to achieve more complex goals, mainly because of its deterministic approach and the lack of representation for more complex views, which may include temporal constraints, conditioned actions and uncertainty. Plan space planning, first suggested by Sacerdoti & Center (1975), provides a more complete view of a planning problem by introducing the notion of partially specified plans.

In a plan-space graph, nodes are partially specified plans instead of sets of conditions that represent states of the world. Arcs in the graph represent plan refinement operations intended to further complete a partial plan. Using this notion of a plan space graph, a planner starts from an initial node, corresponding to an empty plan, and searches for a final node containing a solution plan that achieves the specified goals. Hence, plan space planning does not merely generate action sequences, as it explicitly considers two different operations: choosing actions to be executed and define the ordering in which those actions are to be organised in order to achieve the goal.

A partial plan[1] can be viewed as a structured collection of actions that provides their causal relationships, as well as their intrinsic ordering and variable binding constraints. A partial plan is no longer partial when no open conditions (conditions not yet satisfied) exist, all actions are totally ordered (including actions that can occur in parallel) and all binding constraints of variables are consistent, that is, it is a complete and consistent solution plan that defines a path from the initial state to a state containing all goal propositions.

Although plan space planning allows solving more interesting and complex problems, it lacks however a notion of explicit states along a plan, which makes it difficult to use domain specific heuristics to improve the efficiency of the search space, thus compromising the scalability of this approach. Nevertheless, plan space planning still presents some important advantages. Building partially ordered and partially instantiated plans provides more flexibility when it comes to control the execution phase, as the flexibility of this approach allows detecting and resolving flaws in the plan efficiently. This is due to the fact that the causality representation used in the partial plans allows having an explicit notion of which operator has caused which result in a plan, thus helping solve flaws that may occur.

### 2.1.3 Planning Graphs

Both state-space and plan-space planning present some compelling advantages. While state-space planners deal with a simple abstract view of states, which allows

[1] A formal representation is given in (Ghallab *et al.*, 2004)

them to provide a plan as a sequence of actions, plan-space planners synthesise a plan as a partially ordered set of actions, opening the planning process to more complex problems.

The *planning-graph* (Blum & Furst, 1997) approach takes the best of both worlds. On one hand, it allows having a clear representation of states and to make use of domain-based heuristics much like in state-space planning. On the other hand, the condensed and parallel graph representation of actions and propositions provides a clear sense of causality between them, thus allowing to easily detect incompatibilities amongst actions and propositions as efficiently as in plan-space planning.

Regarding the actual representation, the *planning-graph* adopts a solution somewhere in the middle by providing a sequence of sets of actions, which represents all sequences starting with a set of actions that can be executed in any order, followed by another set of actions that can be executed in any order, and so forth. More precisely, given the initial and goal states and a set of potential actions, a *planning-graph* consists of a directed, levelled graph where levels alternate between proposition levels containing proposition nodes and action levels containing action nodes. The first level is a proposition level composed of proposition nodes corresponding to all of the propositions of the initial state. The second level is an action level composed of action nodes, one for each action whose preconditions are satisfied by the propositions in the first level. The third level is a proposition level composed of proposition nodes which represent the propositions created by the effects of the actions in the second level and by the propositions created by previous proposition levels (also referred to as *no-ops*).

The *planning-graph* is built this way until a proposition level is reached where all propositions of the goal state are included. Arcs (or edges) in a planning graph represent relations between actions and propositions, where action nodes in an action level are connected by precondition-arcs to their preconditions in the previous proposition level, by add-arcs to their add-effects in the next proposition level, and by delete-arcs to their delete-effects in the next proposition level. The extraction of a plan from a planning graph is similar to searching an *And/Or* graph, where *Or*-branches of a proposition are arcs from all actions in the preceding action level that satisfy this proposition, and *And*-branches from an action

node are its precondition arcs.

A *planning-graph* does not represent a valid plan for a planning problem. Instead, it uses the principles of independence and mutual exclusion to drastically reduce the search space and help finding a valid plan faster. An action is considered independent from another action, if the effects of the former do not interfere with the preconditions and effects of the latter. Independent actions can be arranged in a plan in any order (or in parallel if there are multiple executing agents) with exactly the same outcome, hence, the output of a planning graph being a sequence of sets of actions. Two actions at a given action level are mutually exclusive (also referred to as being *mutex*) if no valid plan could possibly contain both. Similarly, two propositions at the same given proposition level are mutually exclusive if no valid plan could possibly make both true.

*Graphplan* (Blum & Furst, 1997) is an example of a planning algorithm that relies on a *planning-graph* structure. The *Graphplan* algorithm iteratively expands the *planning-graph* by one level and then searches backwards from the last level of this graph for a solution. However, the initial iterative expansion process is done until a proposition level is reached where all goal propositions are included and no pairs of them are *mutex*. This is done because it does not make sense to start searching for a plan in a graph that has not reached the goal state yet. The search procedure then looks for a sequence of non-*mutex* actions that achieve the goal propositions. Preconditions of the chosen actions become the new goal propositions and the process continues. A failure to meet the goal at some level `i` leads to backtrack over all other subsets of actions in level `i`+1. If the first level is successfully reached, then the corresponding sequence is a solution plan. This iterative graph expansion and search processes are pursued until either a plan is found or the search reveals that no solution can be found in the *planning-graph*.

*Graphplan* has revolutionised automated planning research mainly because of its simple, elegant algorithm and its representation of planning problems that created the basis for an extremely fast planner (Weld, 1999). Some work has been done on extending *Graphplan*, namely to handle actions with conditional effects (Kambhampati *et al.*, 1997) (Anderson *et al.*, 1998) and to handle uncertainty and sensing actions (Weld *et al.*, 1998). Nevertheless, these extensions apply

to the one agent planning paradigm and do not explore the potential of using *Graphplan* in a distributed setting.

## 2.2 Multi-agent Planning and Coordination

Distributed Planning is a specific area of Distributed Artificial Intelligence, aimed at developing mechanisms that allow finding solution plans while coordinating the activities of multiple intelligent agents. The motivation for this branch of automated planning came from the need to solve more computationally complex problems, which could not be solved by centralised planners such as *Graphplan*. Parallelism, cooperation and concurrency within multi-agent systems have since then been the main areas explored in Distributed Planning research.

Distributed Planning is carried out by intelligent agents, which are autonomous entities perceiving and acting upon the environment. An agent is able to communicate with other agents in order to achieve individual or shared goals. This abstraction of an autonomous entity allowed researchers to have a different view of Artificial Intelligence, namely as a network of intelligent nodes that can interact to further extend their capabilities, as opposed to a single-agent view that is responsible for performing all the desired tasks in an environment. Hence, a multi agent system can be defined as a loosely coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver (Durfee & Lesser, 1989).

As stated before, taking advantage of the decentralised control of such distributed environments requires that coordination mechanisms exist that are able to avoid conflicts that arise from the concurrent interactions of agents, which otherwise would result in a turmoil. Considering this necessity, we can divide the distributed planning process into five separate activities that may occur in this or in a different sequence depending on the taken approach:

- *Goal/Task decomposition* – the agents refine the initial goal/task such that each created subgoal/task matches one or more of an agents capabilities;
- *Subgoal/task allocation* – the agents attempt to assign subgoals/tasks to each other according to the matching process performed in the first stage;

- *Individual planning* – each agent tries to find a plan to solve the subgoal allocated in the previous stage;
- *Overall coordination* – the agents coordinate their activities in order to ensure that all of them are working to achieve the global goal;
- *Plan execution* – the agents execute the jointly built plan in a coordinated fashion.

Although this decomposition is convenient to explain the problem of multi-agent planning, most researchers have addressed the problem from different points of view, which include focusing only on one step, interlacing some of the steps or offering alternatives as how to perform one single step. With that in mind we have approached this analysis from a different perspective, dividing the review of related work into five different categories that are transversal to the list depicted above.

In sub-section 2.2.1, we review research that mostly covers the goal decomposition and allocation stages. Sub-section 2.2.2 focuses on pre-planning coordination, a type of coordination that is performed prior to the act of planning to ensure that agents will not compromise to actions that may affect the activity of other agents. Sub-section 2.2.3 is dedicated to interleaved planning and coordination, in which both activities are performed in an iterative process to maintain the consistency of the distributed problem solving. Sub-section 2.2.4 reviews work that focuses on post-planning coordination, in which agents are left free to perform individual planning and then merge their solution plans into a unified global solution. In subsection 2.2.5, we describe some approaches that perform the coordination process at the final stage, the execution of the solution plan.

In spite of the variety of approaches, domains and contexts, most of the research work on this field presents the same limitations: they rely on some sort of centralised component or on a pre-defined structure/knowledge that rules the activity of all entities in the environment, thus compromising scalability and robustness. Scalability can be compromised when a system fails to keep up with the growth rate of the environment due to either centralisation of some components or the use of communication intensive techniques. We consider that systems are

not robust when they are not able to effectively deal with constant changes and unexpected events whether they are caused by failures in planning or execution, or simply by dynamic environments where agents and capabilities are added or removed without notice.

### 2.2.1 Goal/Task Decomposition and Allocation

The Hierarchical Task Networks (HTN) approach (Erol *et al.*, 1995) (Erol, 1996) is a paradigm for task refinement in coordination environments. HTN representations are based on actions and states of the world similar to those used in state-space planning (see section 2.1.1). However, HTN planning is different from state-space planning in the sense that the objective is not to find a sequence of actions that will bring the world to a state that satisfies certain conditions. Instead, HTN planning searches for plans that accomplish task networks. A task network is a collection of tasks that need to be carried out, together with constraints on the order in which tasks can be performed, the way variables are instantiated, and on the literals that must be true before or after each task is performed (Erol *et al.*, 1995). The goal of an HTN planner is to decompose the task network into primitive tasks (process which can be done through the application of methods) and find a conflict free plan that can execute the tasks in the task network. A method is a syntactic construct that states how a task can be achieved by representing the way it can be decomposed.

The decomposition of task networks into primitive tasks, using an HTN based approach, requires the complete knowledge of the available planning operators. This has often been addressed by using centralised views of the planning space (Amigoni *et al.*, 2005), such as directories or brokers where agents can register their operators. However, these approaches are not scalable and central points of failure or bottlenecks compromise the robustness or the efficiency of such systems. Even though there have been some attempts to distribute the HTN planning approach, namely through the decomposition of goals into team (sub) plans and individual plans (Bonnet-Torres & Tessier, 2005), this paradigm still needs task networks and methods to be provided *a priori* (often by a human user). This compromises the application of HTN-based approaches in highly dynamic environments where no agent knows all the capabilities of all other agents.

From a goal allocation point of view, most approaches rely on centralised components, such as blackboards (Wellman, 1993) (Wellman, 1996) (Walsh, 1999), tables of capabilities (Fung & Chen, 2005) or broker-like auctioneers (Walsh *et al.*, 2000) (Wellman *et al.*, 2001). An obvious limitation of centralised approaches for task allocation in very large networks is its non-scalability.

To avoid depending on centralised components, the Contract-Net protocol (Davis & Smith, 1983) can be used by a manager agent to broadcast bid requests (announcements of sub-tasks that need to be performed) in a network of potential contractor agents. This protocol enables dynamic task allocation, allows agents to bid for multiple tasks at a time, and provides natural load balancing (busy agents do not need to bid). It does not, however, detect or resolve conflicts, there is no pre-emption in task execution (time critical tasks may not be attended to), and it is communication intensive (Jennings *et al.*, 1998). In fact, the Contract-Net protocol relies on the existence of capabilities tables (Durfee, 1999) to help the manager determine where the requests for bids should be sent. When such tables are not available, the manager uses non scalable and uninformed *flooding* techniques (see section 2.3.1) to broadcast the request to all agents, which causes the congestion of the network.

Other approaches (de Weerdt *et al.*, 2007) proposed the use of social networks of agents for solving the task allocation problem. These social networks help agents choose the agents that should be allocated to each task. Unfortunately, the structure of the social network is imposed on the agents by some organisational based method, instead of being built dynamically. Thus, this process limits the range of applicability of the approach because it does not allow the dynamic evolution and adaptation of the social network.

### 2.2.2 Pre-planning Coordination

In this approach, pre-imposed rules or organisational structures (Abdallah & Lesser, 2004) (Jamali & Zhao, 2005a) (Gaston & Desjardins, 2005) define the way the agents in the society interact and operate in the environment. With this kind of implicit coordination approach, agents follow local rules of behaviour that ensure that they can operate without having to worry about interference from other agents (de Weerdt *et al.*, 2005).

Social laws (Shoham & Tennenholtz, 1992), as generally accepted conventions that each agent has to follow, were the first proposal to address this issue. They ensure that, once an agent adopts a goal, no other agent would interfere. One of the best real-world examples of this kind of technique is the application of traffic rules. Little or no communication at all is required between drivers as long as they all respect traffic signs and rules. Even though this technique can be very efficient from the coordination point of view, it requires that social laws exist and are known to all entities participating in the society.

Some authors (Ephrati *et al.*, 1995) believe that social laws are typically very difficult to design and are very complex. Instead, they propose a different approach for pre-planning coordination. By using a filtering mechanism where they bypass options that are incompatible with any agents known or presumed goals, agents are expected to improve their performance in a society, but not to guarantee success for each specific goal. The filtering strategy can be generated straightforwardly from the abstract properties of the environment and the interaction (Ephrati *et al.*, 1995). This contrasts with social laws, which are imposed on the environment where agents operate. Nevertheless, this filtering technique requires that agents have at least partial knowledge of the goals and intentions of other relevant agents in order to avoid conflicting with them. Hence, agents need to engage in communication intensive interactions to collect this knowledge from other agents in the network.

### 2.2.3 Interleaved Planning and Coordination

Since communication among agents in a distributed problem solving environment is a major issue regarding the systems scalability, an alternative approach is to explore a coordination strategy based only on the observation of the environment. The Multi-Agent Planning System (*MAPS*) (Tews & Wyeth, 2000) operates without explicit communication between agents, relying upon observation of team members to produce meaningful coordinated behaviour in the domain of Robot Soccer. By generating an abstract representation of each agents environment at a particular point in time, (*MAPS*) allows the agent to observe important objects and behaviours of other agents before setting new goals, choosing their

actions and planning future moves. This kind of approach is suitable for spatial environments in which it is possible to observe the behaviour of other entities, such as in robots coordination systems. Although observation based coordination allows agents to prevent conflicts while acting according to observations they make of the environment, it often leads to far optimal solutions since they reason on their future intentions based on actions that already took place (executed by other agents) and cannot be undone (or being undone contributes to increase even more the size of the plan). In contrast, communication-based coordination reaches optimal solutions more easily but at the cost of possibly limiting scalability because of the excessive communication involved in the coordination process.

Adopting a *divide and conquer* strategy to solve the multi agent coordination problem has proven to be an effective alternative. Cox *et al.* (2003) propose the use of goal transformations as a coordination mechanism. Basically, this strategy allows an agent, who has to solve goal `G`, to solve a goal `G'` instead that generates a sub-solution and then pass the remainder of the goal (i.e. `G` minus `G'`) to another agent. In more detail, an agent looking to solve goal `G` must solve the set of open conditions of this goal, *i.e.*, it must have the necessary operators to solve all conditions that are not true in the desired goal state. If the agent does not have the necessary skills to do so, it divides the open conditions into two sets: one with the set of open conditions for which it has an operator; and another with the remaining conditions. It is this set of open conditions, for which the agent cannot contribute, that is sent to another agent, hoping it will contribute to satisfy them.

In order for the agent to know where to delegate this set of conditions, this approach considers each agent as a sub-domain, which contains not only those operators assigned to it, but also a set of phantom operators that are not assigned. A domain can be split into any number of sub-domains bounded by the total number of operators in the domain. A phantom operator points to the agent(s) to whom the operator is assigned. This way, communication between agents is encoded in every sub-domain. Hence, if an agent does not own an operator, it knows which agent to enlist for that operator. From a network topology point of view, this means that each agent must have a phantom connection to all

other agents in the network (which in network topology is referred to as a fully-connected network), which is prohibitive for very large and dynamic networks with high churn rates (the rate at which agents enter or leave a network) making the approach non-scalable. In the experiments presented in this paper, the authors have only used a maximum of 3 agents with a maximum of 3 operators each. The described mechanism concatenates the resultant sub-plans from agents into a final solution plan, without a central coordination process. However, it does not take into account possible conflicts that may arise from the fact that agents only contribute to parts of the problem without considering the effect that their decisions may have on other agents' contributions.

### 2.2.4 Post-planning Coordination

On one hand, defining a set of rules a priori to prevent conflicts between agents is impaired by the difficulty to perceive all possible interdependencies between the agents and their operations. On the other hand, while coordinating and planning at the same time may facilitate the task of detecting conflicts, resolving them while providing an optimal solution plan can be a very lengthy process. With this in mind, some researchers decided to explore the post planning approach to multi agent coordination. This approach has the advantage of making use of the parallel processing power of a multi agent system for the planning process, while postponing the coordination process until after all agents have made their contributions.

Georgeff (1988) described a centralised method for synthesising multi agent plans from simple single agent plans. The author proposes inserting communication acts into the single agent plans so that agents can synchronise their activities and avoid harmful interactions. The method performs an interaction and safety analysis to identify critical regions in the plans. This method allows inserting communication primitives into the plans and creating a supervisor process that will handle execution synchronisation.

An alternative (also centralised) approach was described in (Ephrati & Rosenschein, 1994), which integrates sub plans provided by single agents (solving sub goals of a major desired goal) into a joint, multi agent plan. Their algorithm

performs an $A*$ search over possible combinations of the plan steps in the sub plans to arrive at a near globally optimal solution. However, this approach is incomplete, i.e., it is not guaranteed to return a solution to a given coordination problem (Cox & Durfee, 2005), even if a solution exists.

A centralised approach for detecting temporal conflicts between plans was proposed in (Tsamardinos *et al.*, 2000). The authors propose constructing a conditional simple temporal network to identify temporal conflicts between individual plans. One of the problems of this kind of plan merging approach is the possibility of creating cyclic interdependencies between agents that will lead to deadlocks (de Weerdt *et al.*, 2005).

Cox & Durfee (2005) also presented a centralised post-planning coordination algorithm. The coordination process consists of merging the plans of the multiple agents, thus centring its activity on detecting plan step merge flaws (steps whose post-conditions subsume all of the necessary post conditions of another step). The described coordination algorithm starts by analysing a flawed (or inconsistent) multi agent plan and detecting individual flaws that need to be repaired. While repairing flaws, the algorithm needs to iteratively analyse the new plan to check if flaws still exist or if new flaws were not created by the repairing process. The algorithm also maintains a record of the best solution seen so far to make sure that it returns the best consistent plan. In order to avoid cyclic merged plans, which is not done in (Tsamardinos *et al.*, 2000), a cycle check is done by performing a *depth-first* search on the partial order of the steps. If a particular step is visited more than once, then the plan cannot be used since it is cyclic.

All of the presented approaches suffer from being based on a centralised or communication-intensive strategy for plan merging, which is usually computationally expensive. This can compromise the scalability of the system if the number of agents (and thus, the number of individual plans to be merged) is too high. In (Scerri *et al.*, 2007) a decentralised coordination approach for large networks of autonomous vehicles is described. This approach is neither computationally expensive nor communication intensive because it is based on the assumption that, in sparse environments, collisions are rare. This allows vehicles to plan independently and then resolve the small number of conflicts that actually occur. The coordination algorithm basically operates by having each

vehicle send their planned paths to close by teammates. Each vehicle is then required to check for conflicting paths that they have been informed about and inform those involved when any conflict is detected. Unfortunately, operating in this kind of environments has some particular characteristics that cannot be mapped to other domains, such as detecting close by entities. In non geographic domains, detecting close by agents would be equivalent to an agent being able to detect which agents are producing plans that can potentially conflict with the plan that it is producing. This can only be done if all agents know each other in a network, which would lead to fully connected networks. Unfortunately, as previously mentioned, fully connected networks are not scalable.

### 2.2.5 Execution and Coordination

In distributed problem solving, multiple agents collaborate to build a plan with the necessary steps that need to be taken in order to achieve a certain goal. It is only during the execution phase that the steps actually change the world. Different strategies can be used to execute the plans produced by the agents in a network, but seldom coordination activities are dealt with only at the execution stage. Possible exceptions are observation based coordination approaches (Tews & Wyeth, 2000), which rely on the observation of previously executed plans to trigger a new planning phase. Instead, the execution stage is used only to carry out the plan as instructed and learn something from the experience to improve future planning and execution processes; much like when a person follows a "*recipe*" and learns that there is an error or there is room for improvement in some steps of the "*recipe*". The motivation behind this kind of approach is the unexpected behaviour that may occur after the execution of the plan, which could not have been predicted in the planning stage.

Sugawara *et al.* (2004) have proposed a learning method in which agents explore previously used plans to improve problem solving in environments where similar problems appear repeatedly. Lopes & Botelho (2007) have proposed an alternative approach that relies instead on (either past or current) context information to improve the execution process of compound semantic web services. This approach is based on a special-purpose broker agent, the Service Execution

Agent (SEA), which is able to receive descriptions of compound semantic web services, isolate the atomic steps involved in the plan control constructs and then request the execution of each atomic step to the appropriate service provider. While doing so, SEA collects relevant context information (such as the service providers requests queue and average execution time) from a generic context system (Costa *et al.*, 2008) to adapt and improve the service execution process.

Although these approaches may help improve the execution process and may even help improve future planning, they reflect a separation between the planning and execution process that does not contribute to efficiently address the overall coordination problem. Some researchers suggest that the planning, execution and coordination processes should be interleaved as if they were only one process. Distributed Continual Planning (Desjardins *et al.*, 1999) is a paradigm that consists of interleaving planning and execution into a continual process which takes into account changes in the environment and handles them through re-planning and employing effective multi agent coordination techniques based on the success or failure of previous decisions.

As described throughout section 2.2, many proposals have been made to address the issues pointed out by this paradigm. However, to this date, no single generic approach has proved to be able to effectively coordinate large networks of autonomous intelligent agents in different domains of distributed problem solving. Main reasons for this are the centralisation of some components of the environment or the use of communication intensive techniques; and systems that are not able to effectively deal with constant changes and unexpected events whether they are caused by failures in planning or execution, or simply by dynamic environments where entities are added or removed without notice.

## 2.3 Peer-to-peer Computing

Cooperating in distributed problem solving processes requires agents to be able to discover other agents to delegate the sub-problems for which they cannot contribute. Ideally these other agents should be as adequate as possible to solve the delegated sub-problems. In small networks, the best strategy is for an agent to know the skills of all other agents, thus allowing it to easily determine where the

sub-problem should be forwarded. However, in large-scale dynamic networks, this kind of approach is unacceptable, as it is very difficult for an agent to maintain an accurate and up-to-date view of the entire network and to know exactly where all agents (and corresponding capabilities) are at a certain moment in time.

It is clear that the approaches described in section 2.2 are not suitable to be used in highly dynamic large networks of intelligent problem solvers, as most of them present scalability and robustness issues. With this in mind, we believe the use of P2P computing techniques can improve the way multi agent coordination is done. Indeed, some approaches have shown that P2P network search mechanisms and techniques have helped multi agent systems to become more scalable and robust.

### 2.3.1 Network Structure and Search Mechanisms

The research on P2P computing classifies P2P systems according to two dimensions: network structure and search mechanisms. Network structure refers to the existence of some sort of structure according to which, some peers have different responsibilities or are hierarchically organised within the network. In terms of network structure, P2P systems can either be *pure* (also referred to as *unstructured*) or *hybrid* (also referred to as *structured*). In pure P2P systems, all peers are equal in responsibilities and no hierarchy exists, whereas in *hybrid* P2P systems, peers are organised in specific hierarchies or some peers – also referred to as *super-peers* or *ultra-peers* – have different responsibilities. *Hybrid* P2P systems also refer to networks where peers are connected according to a specific structure based on the resources they manage.

The search mechanism dimension classifies P2P systems according to the way peers search other peers or specific resources in the network. According to this classification, P2P systems can employ *uninformed* searches (also referred to as *blind* searches) or *informed* searches. In an *uninformed* search, each peer searches the network by randomly querying other peers, whereas in an *informed* search, each peer uses additional information about other peers resources to select the peers that will be contacted during the search process (Bianchini *et al.*, 2006).

In unstructured networks, where peers cannot rely on any information to optimise the search process, searching a certain network resource or peer is often

carried out by a *flooding* algorithm or a *random walk* algorithm. In the *flooding* algorithm (also referred to as *breadth-first* search), a peer broadcasts the search query to all of its neighbours, which in turn will apply the same process until the search result is found or some condition holds. A *Time-To-Live* (*TTL*) constant is often used to stop the *flooding* propagation at a certain level. In the *random walk* algorithm (also referred to as *k-random* or *depth first search* – when `k` = 1), a peer chooses a `k` number of random neighbours to propagate the search query. These, in turn, will use the same process until the search result is found. Both algorithms present some disadvantages. *Flooding* increases network load with copies of the query message but may retrieve the results faster, whereas a *random walk* reduces the network load but increases search latency.

In recent years, some approaches designed search mechanisms based on some variations of these two algorithms. Iterative deepening (Yang & Garcia-Molina, 2002) is an example of an effort to improve the use of *flooding* techniques. A peer, employing this search mechanism, initiates multiple *breadth-first* searches, over the iterations of the technique, with successively larger depth limits, until either the query is satisfied, or a maximum depth limit has been reached. To avoid having nodes processing the same request multiple times, *Resend* messages are used to guarantee that only nodes beyond the previous depth limit process the request; nodes within the previous depth limit only forward the request.

### 2.3.2 Informed Searches

In an attempt to improve the effectiveness of search mechanisms in P2P networks, *informed* searches were introduced, offering the possibility to improve the performance of the discovery process by using information on peers and their resources. This information is obtained from previous queries. Knowing exactly which peers to use when propagating a query can help reduce the network load (less *flooding*) while improving the search performance.

*Routing Indices* (Crespo & Garcia-Molina, 2002) allow nodes to forward queries to a subset of neighbours that are the best candidates to satisfy the query. The subset of candidate neighbours is identified by evaluating an index table that contains the inventory of the neighbouring nodes (Ratsimor *et al.*, 2004). This

approach is based on a push update technique where each peer sends to its neighbouring nodes information about its resources and constantly updates them whenever its resources change. Similar approaches are exploited in the *Directed Breadth First Search* (Yang & Garcia-Molina, 2002) and in the *Intelligent Search* mechanism (Kalogeraki *et al.*, 2002) where each peer in the network builds a profile of its peers and uses the profile to determine the peers that are more likely to answer each query. Using *routing indices*-like approaches enables agents to effectively answer queries as they gather information on the interests and information provision abilities of others, without altering the topology or imposing an overlay structure to the network of acquaintances (Vouros, 2007), as long as the number of agents does not increase to very large numbers (tenths of thousands or higher).

A self-learning approach is the basis of the *Adaptive Probabilistic Search* (Tsoumakos & Roussopoulos, 2003), where each peer uses feedback from previous searches to adjust the probability of successfully using certain neighbouring peers in future searches. This approach constitutes an advantage over the ones proposed in (Yang & Garcia-Molina, 2002), (Crespo & Garcia-Molina, 2002) and (Kalogeraki *et al.*, 2002) because it does not introduce an excessive overhead to update the indexes at the neighbours. A more flexible feedback-based approach is employed by the *Directed Searches* (Lv *et al.*, 2002), where peers use a vast set of metrics, which range from the number of successfully returned query results to network connectivity and latency, to learn from previous interactions and improve future searches.

One approach used to improve the *uninformed* search mechanisms in unstructured P2P networks, described above, was based on the use of indexes and statistical information to help peers choose the appropriate neighbours to which future search queries should be routed. Another approach is to introduce some sort of structure to improve message routing, which is usually done by partitioning the network into a set of communicating clusters of peers that are connected amongst them by a network of super peers (Bianchini *et al.*, 2006).

A *super-peer* belongs to a higher level of a peers hierarchy, which is usually based on content related criteria. *Super-peers* are responsible for managing and facilitating search processes among the peers in their clusters (by maintaining an

index of their peers resources) and for communicating with neighbouring *super-peers* to further extend search processes that could not be resolved locally. An example of this structured approach was introduced in the *FastTrack* P2P platform (Liang *et al.*, 2006). Hierarchical approaches such as the ones based on these special-purpose peers come at the expense of potential semi catastrophic failures of *super-peers* near the top of the hierarchy (Balakrishnan *et al.*, 2003).

In order to offer a scalable and yet robust infrastructure for P2P networks, an alternative approach, based on *Distributed Hash Tables* (*DHT*), has been proposed. *Chord* (Stoica *et al.*, 2001), *Pastry* (Rowstron & Druschel, 2001) and *Tapestry* (Zhao *et al.*, 2001) are examples of *DHT* implementations. The *DHT* approach is based on the sole principle that a resource can be identified by a numeric key that is created through a hash function, based on the resources contents. In order for a resource to be published under a specific key, the peer routes the publishing request to the peer with the key closest to the resources key (based on some *closeness* function), which in turn stores that information in a routing table. When a peer searches a specific resource in the network, it routes the request to the peer with the closest key, which in turn will apply the same process until the resource is located in the network. The *Content Addressable Network* (Ratnasamy *et al.*, 2001) differs from these approaches by operating in a multi dimensional view of the *DHT*s, that is, by allowing for peers to search for resources in the network using more than one type of key simultaneously.

Unfortunately, without some concrete way to describe relationships between resources, these approaches do not leverage the potential of semantically linked peers to improve the resource coordination process. Semantic links aim at providing a more meaningful way to connect peers and their resources, thus allowing for peers to easily combine their resources with other semantically related peers.

## 2.4 Hybrid Approaches

The use of Multi-Agent Systems (MAS) to efficiently coordinate resources in collaborative environments has gained some attention, in part, due to the advances in P2P computing. The evolution of search mechanisms, which were showing signs of scalability and robustness, paved the way for the development of more

complex and intelligent systems. However, current search algorithms do not ensure, by themselves, that a suitable basis for agents to cooperate in problem solving is created.

Using the semantic link paradigm to create meaningful connections amongst the agents in an environment (also referred to as *semantic overlay networks* (Crespo & Garcia-Molina, 2005)) may be an effective way to optimise the coordination process. If, for example, a peer manages a resource that is somehow related to another resource that is managed by another peer, then it is important that a semantic based connection exists between these two peers stating the meaning of their relationship. This semantic link can then be used to improve future searches or collaboration initiatives. Using P2P computing and semantic descriptions of web services, several approaches have addressed agent/service coordination issues especially related to matchmaking, discovery, planning and composition.

A decentralised web service organisation approach is presented in (Yu *et al.*, 2004), in which a DHT-based catalogue service is used to store the semantic indexes for direct service publication and discovery. This semantic indexation consists of a classification of the services based on domain-related categories. A similar approach was described in (Jin *et al.*, 2005), where peers in a network advertise their "*service expertise*" based on domain categories. The algorithm used to spread the advertisements within the P2P network is based on a ranking system, which allows peers to route their "*service expertise*" only to peers that operate in similar domain categories (according to a similarity function).

GloServ (Arabshian & Schulzrinne, 2007) uses a keyword based taxonomy search on a hierarchical hybrid P2P network to build a semantic overlay between the peers that operate in the same (sub) domain. Several other keyword based mechanisms for semantic web services discovery and matchmaking on P2P networks that do not rely on centralised taxonomies or domain categories were proposed. The keyword search in these approaches is done at the level of operation names (Liu & Zhuge, 2005) or non-functional service descriptions (Toma *et al.*, 2005) (Sapkota *et al.*, 2005).

The Web Services Peer-to-Peer Discovery Service (WSPDS) (Banaei-kashani *et al.*, 2004) is a service discovery approach in pure P2P networks, where semantic links between peers reflect the similarity of the services they provide.

The matchmaking process is carried out by comparing services' inputs and outputs. A similar matchmaking process is suggested in Bibster (Haase *et al.*, 2004), where peers' capabilities are semantically linked by first applying the same inputs and outputs comparison as in WSPDS and then by ranking services through a similarity-based expertise matching. The METEOR-S Web Service Discovery Infrastructure (Verma *et al.*, 2005) presents a similar approach to the WSPDS and Bibster but it relies on a hybrid P2P network architecture where special peers are introduced to handle a global ontology. The approach presented in (Romeikat & Bauer, 2007) also uses semantic matching at the level of inputs and outputs but it differs from related approaches by using a DHT based service discovery process on top of a Chord P2P network.

Some systems rely on structured solutions, such as aggregation of peers in communities or the use of middle layers that have specific coordination capabilities. SELF-SERV (Benatallah *et al.*, 2002) is a framework where web services are composed using state charts. The resulting composite services are executed in a decentralised way within the P2P dynamic environment. This framework relies on the concept of service communities (containers of alternative services), which provide abstract descriptions of desired services and allow actual service providers to register in the appropriate community. The distributed execution is managed by coordinator agents that are in charge of initiating, controlling, monitoring and collaborating with their peers to manage the execution of the services they control or provide.

The approach presented in (Küngas & Matskin, 2006) uses a MAS to perform distributed composition of web services, using mediator agents. In (Arpinar *et al.*, 2005), a similar approach is used for automated web service composition over a P2P network, where peers are organised into communities that represent the same domain. The major difference between this approach and (Küngas & Matskin, 2006) is that the former tries to determine links between web services at publishing time (suitable for more stable networks) and the latter does this at composition time (suitable for more dynamic networks). *A-peer* (Li *et al.*, 2003) is a multi-agent-based P2P system where agents rely on hierarchically arranged advertising elements to find the services they need from other agents. This kind of middleware solution is also used in (Ermolayev *et al.*, 2004), which describes

a framework for agent enabled web service composition where an Agent Middle Layer is used to transform service requests into the corresponding tasks in the P2P environment.

Structured systems contribute to optimise the routing mechanisms in P2P computing, however, at the cost of introducing central points of failure and in certain environments, compromising scalability. To avoid these failure prone solutions, some approaches are based on pure P2P networks, such as the inference system presented in (Adjiman *et al.*, 2006). In this approach, each peer can answer queries by reasoning with its local (propositional) theory but can also perform queries to some other peers with which it is semantically related by sharing part of its vocabulary. In order to create these semantic relations (referred by the authors as *acquaintance networks*), new peers joining the P2P system simply declare their acquaintances in the network, that is, the peers they know to be sharing variables with, and they declare the corresponding shared variables. However, the authors do not clearly explain how this "*acquaintances declaration*" process is carried out efficiently in the P2P network.

The study of ant communities has inspired some research on the development of P2P systems based on multi-agent systems. *Anthill* (Babaoglu *et al.*, 2002) is a P2P based MAS which emulates the resource coordination behaviour of ants. In this framework, storage or computational resources (referred to as *nests*) generate requests (referred to as *ants*) in response to user requests. These *ants* travel across the network of *nests* in order to be processed and executed. *Ants* do not communicate directly with each other. Instead, they communicate indirectly by leaving information related to the service they are implementing in the appropriate resource manager found in the visited *nests*. This *pheromone*-like approach, also called *stigmergy*, allows the network to self-organise and improve its performance over time. The idea of assigning agents to carry on requests (*ants*) avoids a non-scalable *flooding* search technique, since each *ant* will only travel to a *nest* at a time and it will not replicate. However, the search performance might be slower because each edge of the network (*nests*) is only travelled once at a time for each request (which is equivalent to a *depth-first* search). The selection of the next *nest* to be visited by an *ant* can either have a deterministic approach (once the network is organised and appropriate overlay networks are available) or

a totally random (*uninformed*) approach. A similar approach to (Babaoglu *et al.*, 2002) is proposed in (Dasgupta, 2005), where mobile agents use *pheromone*-like behaviour to optimise the trails within a P2P network. However, instead of using the update process based on the discovered path, as in (Babaoglu *et al.*, 2002), the mobile agent creates a referral to the query answering node, thus creating a direct link that will improve future similar searches.

A fully distributed approach to the resource discovery problem in a MAS is presented in (Dimakopoulos & Pitoura, 2003). In this system, each agent maintains a limited size local cache in which it keeps information about different resources and the agents that provide them. An agent searching a specific resource consults its local cache and if there is no information regarding the resource, it contacts a `k` random subset of neighbours (to avoid *flooding*), which in turn contact their neighbours. The process goes on until the resource is found in some cache. This system innovates with respect to similar search mechanisms (Yang & Garcia-Molina, 2002) (Crespo & Garcia-Molina, 2002) (Kalogeraki *et al.*, 2002) by proposing the use of inverted caches. Besides maintaining a local cache of agents with certain resources, the agent maintains a cache of agents that have a reference to its own resource in their caches to facilitate the mechanism of updating changes in the network. However, this approach does not address the problem of choosing the appropriate resources that each agent should maintain in its cache. Doing so can help improve search performance in the network over time.

## 2.5 Limitations and Challenges

Our main goal is to build a coordination framework that allows intelligent agents to freely participate in totally decentralised large-scale collaborative environments. In particular, using adequate planning algorithms, these agents would be able to use their own skills to partially contribute to received problems and delegate the remaining (unsolved) parts to other agents that are better equipped to further contribute to the problem.

In section 2.1, we reviewed the classical planning approaches: state-space planning (Fikes & Nilsson, 1971), plan-space planning (Sacerdoti & Center, 1975) and

planning graphs (Blum & Furst, 1997). While these have evolved to address fairly complex problems (Kambhampati *et al.*, 1997) (Anderson *et al.*, 1998) (Weld *et al.*, 1998), they are still inadequate to be used in large distributed environments. Such growing environments need to be efficiently coordinated in order for the collective power of the network to be used in providing solutions to complex problems.

In section 2.2 we have shown that, even though current research is able to address the general problem of coordination for distributed planning, they do so by making use of centralised components (Walsh, 1999) (Fung & Chen, 2005) or organizational-based structures (Shoham & Tennenholtz, 1992) (Ter Mors *et al.*, 2004) (Abdallah & Lesser, 2004) (Jamali & Zhao, 2005a) (Jamali & Zhao, 2005b) (Gaston & Desjardins, 2005) (de Weerdt *et al.*, 2007) that are prone to failures that can compromise the entire system. In fact, major comparisons of multi agent coordination strategies (Ogston & Vassiliadis, 2002) (Ben-Ami & Shehory, 2005) show that centralisation is only suitable when the environment is composed of a few hundred agents and that distributed approaches are clearly more effective for larger networks of agents.

We believe that peer-to-peer (P2P) computing research, which has been focusing on building distributed environments in which peers can seamlessly exchange and share resources between them, may help overcome the issues that have been pointed out earlier.

On one hand, the survey of recent work has led us to conclude that the use of semantic links (Crespo & Garcia-Molina, 2005) may improve the coordination of entities that manage resources in collaborative environments. This improvement can be achieved by using knowledge acquired in previous interactions to enhance future ones, that will rely on more complex dynamically learnt and maintained meaningful connections between agents.

On the other hand, building a semantic overlay network from a set of randomly connected agents requires efficient algorithms that are able to balance search speed and completeness, while allowing the network to evolve and self-organise. Hence, constant dynamic adaptation, network evolution and self-organisation assume very important roles in the development of more robust and scalable intelligent dynamic environments. However, as shown in section 2.3, although P2P

computing does hold this potential, current algorithms and techniques (Rowstron & Druschel, 2001) (Stoica *et al.*, 2001) (Zhao *et al.*, 2001) (Ratnasamy *et al.*, 2001) (Crespo & Garcia-Molina, 2002) (Yang & Garcia-Molina, 2002) (Kalogeraki *et al.*, 2002) (Tsoumakos & Roussopoulos, 2003) (Lv *et al.*, 2002) (Liang *et al.*, 2006) are still not efficient enough to be integrated into multi-agent collaborative environments.

With these limitations in mind, we have established the following challenges to be addressed by our research work:

> Design a discovery mechanism that will enable an agent to find others that are semantically related to it (or its capabilities) or that are more adequate to contribute to solve the yet unsolved parts of a given problem for which the agent cannot contribute, without relying on centralised components or organisation based structures.
>
> Design a distributed planning algorithm that will take into account only partial knowledge of the domain, that is, which allows agents to make partial contributions to a solution plan, considering only the actions of the agent (and possibly the ones of agents semantically related to it).

# Chapter 3

# Testing Scenarios

Our approach is intended to be used in environments where a problem, described as a set of goals to be achieved, must be solved through decomposition and delegation possibly to several agents. In such environments, agents have different capabilities, which may or not be complementary, and it is their collaborative work that ultimately produces a solution to the problem. In this chapter we describe two such environments, the *Rescue Agents* and *Custom Balls Factory* scenarios, in which we have deployed and tested our approach:

- *Rescue Agents* – In this scenario (described in section 3.1), agents represent entities that participate in a rescue operation after the occurrence of a natural disaster, where they have to perform operations such as clearing roads, putting out fires and providing assistance to injured people.
- *Custom Balls Factory* – In this scenario (described in section 3.2), agents represent machines that can apply different types of customisation in the production of sports balls, such as colour, size, shape, fabric type, filing, manufacturing process and other properties.

The scenarios, which were chosen because they represent diverse large classes of coordination problems, are deliberately different to allow analysing and testing different aspects of the coordination approach. On one hand, we have the *Rescue Agents* scenario, which in spite of the low number of different types of entities (paramedics, ambulances, firemen and policemen), is a very complex planning

scenario due to the high level of interaction/cooperation that is needed between the agents. In almost any situation, all entities of the environment are required to intervene to provide the best assistance possible to the injured people, thus making conflicts management the top most priority of the planning activity. Basically, this scenario is intended to represent those coordination problems in which small teams of individuals have to intimately collaborate (to avoid conflicts) to solve very complex or large problems (which usually lead to very large solution plans), such as rescue operations, project planning or robots playing football.

On the other hand, we have the *Custom Balls Factory* scenario, which in spite of involving many different capabilities, is a fairly simple planning scenario. For each manufactured ball, only a very small set of skills is needed from the vast selection of existing capabilities, thus characterising this scenario as a discovery challenge. The planning process on this scenario only becomes relevant when the requested customisation of the ball requires a set of interdependent features requiring a specific execution sequence (for example, a ball must first be fully painted with one colour and only then can stripes be painted with another colour – executing these actions in reverse order would result in the effects of the paint action cancelling the effects of the stripes action). Basically, this scenario represents those coordination environments in which the problems to be solved are usually simple and small but for which the number of possible candidates to participate in the creation of the solution plan is huge, such as service coordination, travel planning or event planning.

The goal of this chapter is to allow the reader to become acquainted with the kind of problems that are addressed by our approach. Through the presentation of these two representative scenarios it will become clear that our coordination approach for distributed problem solving in totally unstructured networks will require two main components:

- A planning algorithm that allows agents to decompose problems into sub-problems and delegate the unsolved parts to other agents;
- A P2P search algorithm that allows the agents to efficiently discover other agents that can solve the sub-problems for which they do not have the necessary skills.

Section 3.3 summarises the presentation of the scenarios and the analysis of requirements for our approach. This analysis, in conjunction with the goals established in Chapter 1 and the conclusions of Chapter 2, constitute the basis of our research work. All listings referenced in this Chapter are in Appendix B.

## 3.1 Rescue Agents

In the *Rescue Agents* scenario, agents represent entities that participate in a rescue operation after the occurrence of a natural disaster, where they have to perform operations such as clearing roads, putting out fires and providing assistance to injured people. This scenario is characterised as having a small number of different entities but with a high degree of complexity due to the high level of necessary interaction/cooperation.

In order to fully understand the requirements of the scenario, we now describe a simplified instance of the typical problem to be solved in this domain from start to finish. To describe the example, we use the Planning Domain Description Language, PDDL (McDermott, 2000), which has become a community standard for the representation and exchange of planning domain models.

Although several different entities could be considered, for this particular example, we will consider only the following:

- *Paramedic* - a medical physician that is able to assist injured people *in loco*, by either providing immediate assistance (such as cardiopulmonary resuscitation) or by diagnosing and dispatching the individual to a hospital;
- *Ambulance Driver* - the ambulance driver (which by association also represents the ambulance entity in the environment) who is able to drive Paramedics, medical equipment/resources and patients from one location to another;
- *Fireman* - an entity responsible for different tasks, such as putting out fires, clearing roads of obstacles and buildings and removing people from the wreckage;

Consider the PDDL code listings B.1, B.2 and B.3 that depict, respectively, the available operators in this domain[1] for *ambulance drivers*, *firemen* and *paramedics*. Each action is composed of its *parameters* (the objects used in the action), *preconditions* (conditions that need to be true prior to the execution of the action) and $effects$ (conditions that are made true after the execution of the action).

According to the action descriptions, both *ambulance drivers* and *firemen* can move from one location to another but a *paramedic* cannot move between locations on his own. A *paramedic* can, however, get in (and out of) an ambulance to then move around through several locations. Also, both *paramedics* and *firemen* have additional operators to, respectively, perform a triage (evaluate the current condition) on injured people and put out fires on specific locations.

Consider the PDDL code listing B.4 and figure 3.1 that depict a typical problem in this domain in which an injured person needs assistance but the only way to reach her is through a path that is blocked by a fire. Also, PDDL code listings B.5, B.6 and B.7 represent, respectively, the initial states for an *ambulance driver*, a *fireman* and a *paramedic* in the example.

In this particular example, we represent the world as a $10 * 10$ grid. Each cell of that grid represents a specific location and it is denoted as $L$ + *row number* + *column number*. For example, $L00$ represents the uppermost left cell of the grid. Dark grey cells represent roads and light grey cells represent other elements of the world that are not relevant for the problem at hand. Each entity in the environment is represented by one or more agents[2]. Agents in the environment that can move from one location to another can only do so through road adjacent cells (excluding diagonals). In this specific problem, an injured person is on location $L84$, an ambulance and a paramedic are both on location $L20$, a fireman is on location $L27$ and there is a fire occurring on location $L64$.

A paramedic can perform a triage on the injured person, but requires an ambulance in order to move to the location where the injured person is. However,

[1]For the sake of simplicity we only show here the subset of the actions of the domain that are required for the problem in the example.

[2]Even though not explicitly represented in the figure, we assume other agents of the same type as the ones presented here also exist and are available to intervene in the problem solving process.

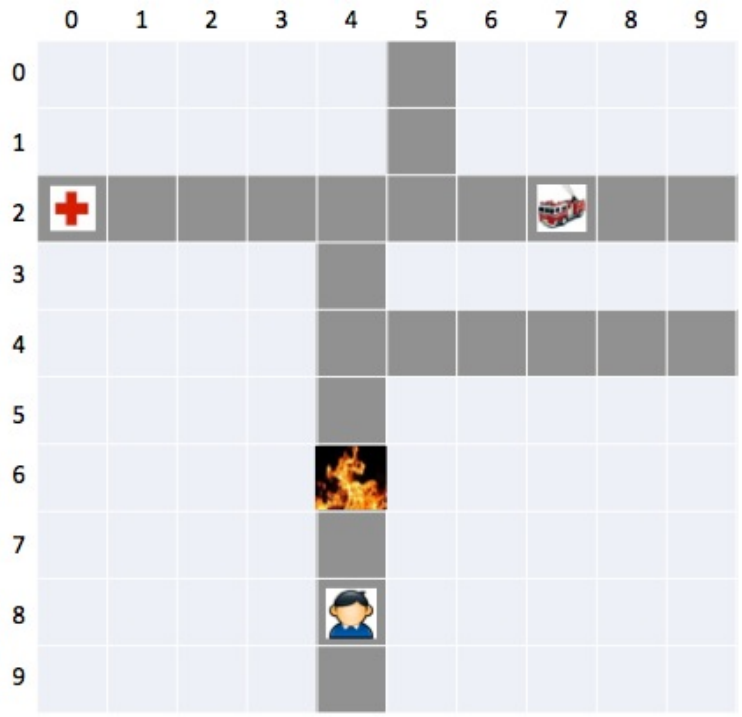


Figure 3.1: *Rescue Agents* scenario problem example

the ambulance cannot access that area while the fire has not been extinguished, task which can only be handled by a fireman. So, this scenario requires the intervention of all three entities described above, but in order to reach the desired goal, they will need to coordinate themselves, that is, determine which of their actions will be necessary and more importantly, the sequence in which they should occur in the final solution plan.

Moreover, the scenario presents some limitations. At boot time, each agent only has knowledge of its own skills (planning operators) and current state and not the ones of others. Each agent is randomly connected to a fixed number of neighbouring agents, with which it can communicate to collaborate on the problem solving process. Also, the agents do not know the problem description until they receive it from another agent that has forwarded it to them.

In this context, an agent that receives the request to solve this problem will analyse it, determine which of its actions can be used to solve part of the problem and then build a partial solution plan. Afterwards, it will communicate with its

neighbours to find other agents, to which it will forward its current partial solution plan. Each agent that receives the partially solved problem will not only become aware of the problem that needs to be solved but also of the contributions that were made so far.

So, let us imagine that for this example, the first agent that receives the problem solving request is a paramedic (`pmedic1`, the one located at $L20$). By analysing the problem[3], agent `pmedic1` determines that it can use its own actions to produce the partial solution plan depicted in Figure 3.2. Grey squares represent actions and the formulas without borders represent propositions. The proposition with Bold font is the goal proposition. Elements in red represent unsolved parts of the solution plan. The elements to the left of the grey vertical line are propositions present in the initial state. Since at this point, the plan is being built by agent `pmedic1`, this initial state includes all propositions of the problem's initial state and the propositions from `pmedic1`'s initial state.

As it is clear in Figure 3.2, the current solution plan requires some sort of action that is able to contribute to the open conditions `(ambulance_at ?a l20)` and `(ambulance_at ?a l84)`. However, agent `pmedic1` does not have any action that can contribute to them. At this point, agent `pmedic1` must initiate a discovery process of some sort to find an agent with the necessary skills. This can be done by using a P2P search algorithm to send a request to neighbouring agents to find an agent with a specific skill that contributes to achieve those open conditions.

Let us assume that `pmedic1` is able to find the ambulance agent `amb1`, whose action `ambulance_move` is just the right skill to complete the solution plan. So, after analysing the current solution plan, agent `amb1` determines that its action can be used to further contribute to the solution plan, as it is described in Figure 3.3.

In this figure, in which we present only the contributions made by agent `amb1` and not the entire solution plan, we have removed some of the steps to make it more legible. The propositions in Bold font represent the open conditions in

[3]At this point, we do not make any assumptions as to which type of planner (or if it is a backward or forward-chaining approach) the agent is using. We simply assume the agent is able to perform a matchmaking process between its actions and open conditions in the current plan.

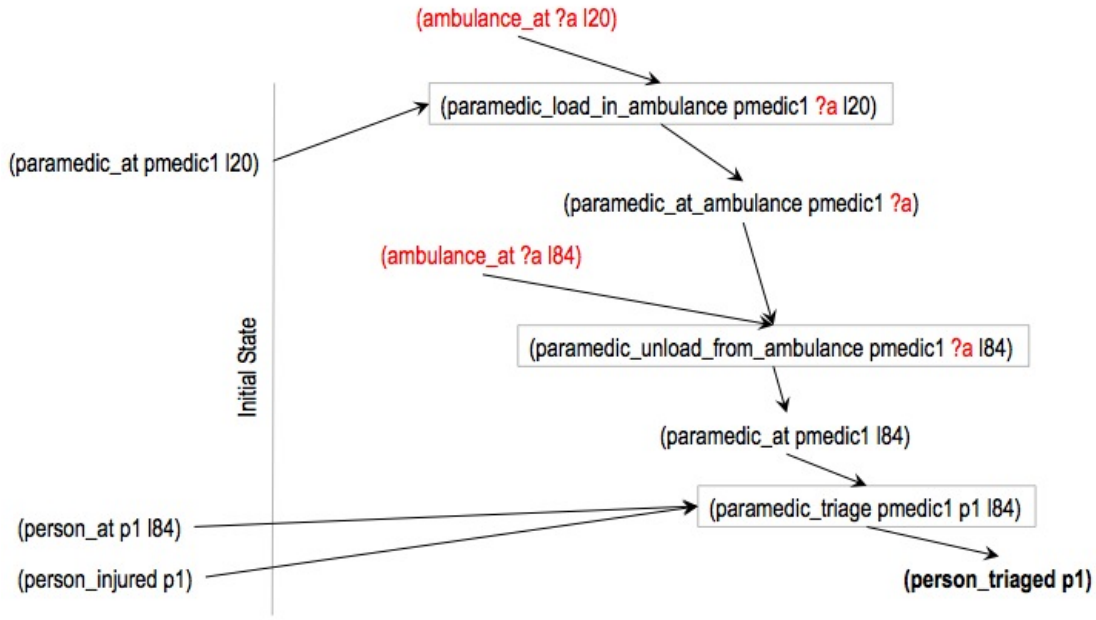


Figure 3.2: Partial solution plan developed by agent `pmedic1`

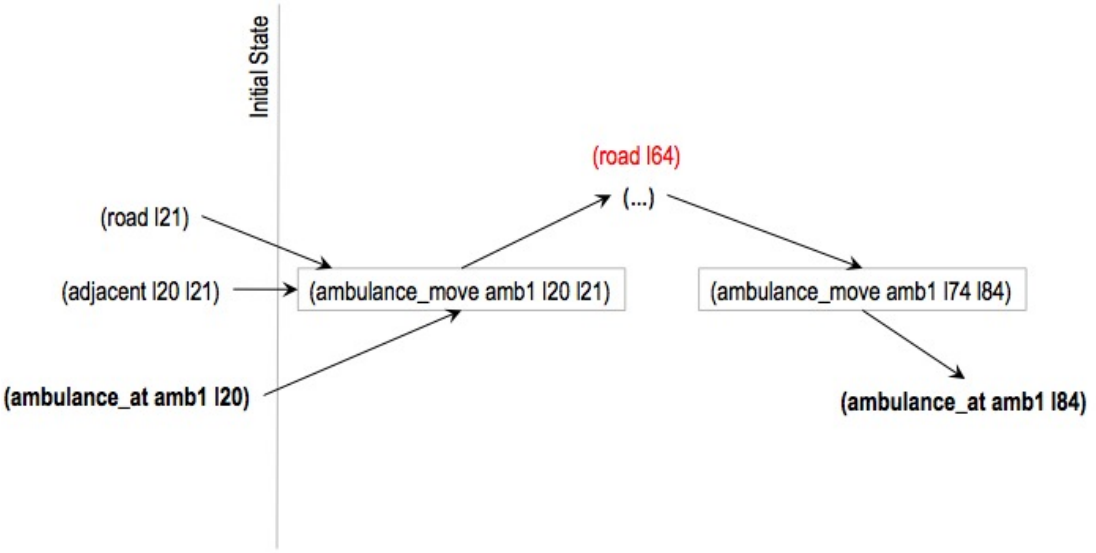


Figure 3.3: Partial solution plan developed by agent `amb1`

agent `pmedic`'s partial solution plan. The figure clearly shows that this solution plan is still incomplete due to the open condition `(road l64)`. The *ambulance* cannot drive through the entire route from `l20` to `l84` because `l64` is on fire, blocking the path between the two locations.

Agent `amb1` needs to engage in the discovery process in the P2P network in order to find an agent with the necessary skills to put out the fire in location `l64`. Agent `fman1` is selected as being one possible candidate. After proper analysis, agent `fman1` makes the contributions depicted in Figure 3.4. By moving to an adjacent position to the fire and then applying action `fireman_putout_fire`, agent `fman1` is able to solve the last open condition of the solution plan, thus producing the final solution plan described in listing 3.1.

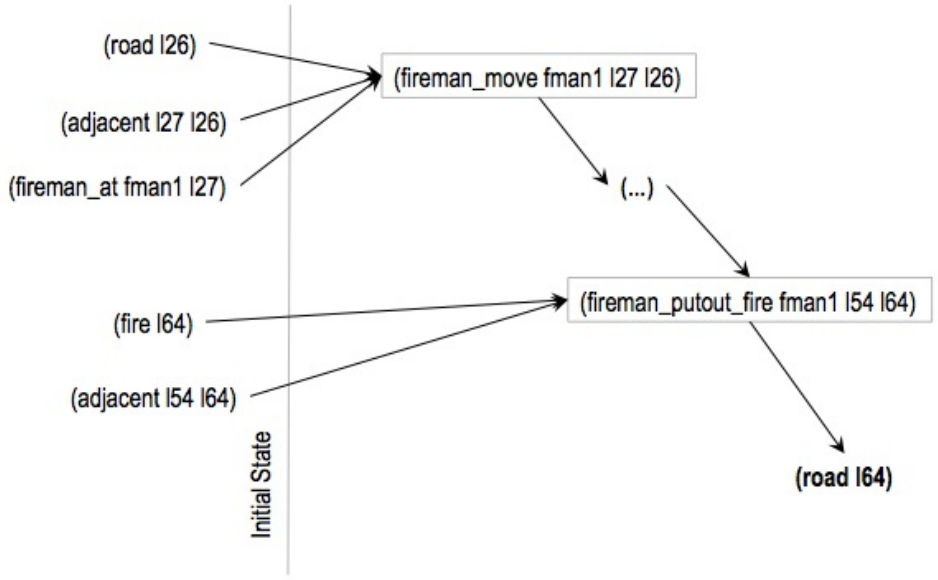


Figure 3.4: Partial solution plan developed by agent `fman1`

The final solution plan is described as a sequence of sets of steps. We use this representation because some actions can occur in parallel since they do not interfere with actions that occur in the same step. In this case, we can see that at the beginning of the plan the paramedic gets inside the ambulance while the fireman is performing its first `move` action. Then, both the fireman and the ambulance execute their `move` actions at the same time. This produces a plan in

which these actions occur in parallel because there is no interference between the actions of the two agents.

```
1  <(fireman_move fman1 l27 l26), (paramedic_load_in_ambulance pmedic1
       amb1 l20)>
2  <(fireman_move fman1 l26 l25), (ambulance_move amb1 l20 l21)>
3  <(fireman_move fman1 l25 l24), (ambulance_move amb1 l21 l22)>
4  <(fireman_move fman1 l24 l34), (ambulance_move amb1 l22 l23)>
5  <(fireman_move fman1 l34 l44), (ambulance_move amb1 l23 l24)>
6  <(fireman_move fman1 l44 l54), (ambulance_move amb1 l24 l34)>
7  <(fireman_putout_fire fman1 l54 l64), (ambulance_move amb1 l34 l44)>
8  <(ambulance_move amb1 l44 l54)>
9  <(ambulance_move amb1 l54 l64)>
10 <(ambulance_move amb1 l64 l74)>
11 <(ambulance_move amb1 l74 l84)>
12 <(paramedic_unload_from_ambulance pmedic1 amb1 l84)>
13 <(paramedic_triage pmedic1 p1 l84)>
```

Listing 3.1: Final solution plan in the *Rescue Agents* domain example

## 3.2 Custom Balls Factory

In the *Custom Balls Factory* scenario, agents represent machines that can apply different customisations in the manufacture process of a sports ball, such as colour, size, shape, fabric type and other properties. This scenario is characterised as having a very large number of different entities but with a low degree of complexity regarding the cooperation between the agents.

Again, we will be showing a complete example of cooperation between several agents in order to fully describe the scenario. There are two main elements in a ball's manufacturing process: the exterior cover and the inflated or filed interior. The exterior cover can either be composed of 4 (or more) slice-like parts or a mix of 32 hexagonal and pentagonal tiles. The interior can either be inflated with air or filed with foam. Naturally, most customisations are applied to the exterior cover.

In this scenario, it is expected that a very large number of different customisations exists. Contrary to the *Rescue Agents* scenario, this scenario is not very

complex from the planning point of view because there are very few conflicts between the different customisations. And, since each customisation is represented by a single agent, it is clear that the scenario constitutes an interesting challenge from the discovery process point of view, instead. That is, most of the time of the problem solving process will be spent finding the agents, amongst so many, which have the necessary skills to apply the requested customisations.

Although many different skills could be considered in this scenario, for this particular example, we will only consider the following agents[4]:

- *trgb_painter* - can paint the top area of a ball in Red, Green or Blue. The painter skill is described in listing B.8 and this agent's initial state is described in listing B.12;
- *bb_painter* - can paint the bottom area of a ball in Blue. The painter skill is described in listing B.8 and this agent's initial state is described in listing B.13;
- *bs_painter* - can paint any part of the ball with blue stripes. The stripes painter skill is described in listing B.9 and this agent's initial state is described in listing B.14;
- *gs_painter* - can paint any part of the ball with green stripes. The stripes painter skill is described in listing B.9 and this agent's initial state is described in listing B.15;
- *assemb* - can assemble the exterior cover and the interior together. The assembling skill is described in listing B.11 and this agent's initial state is described in listing B.16;
- *infla* - can inflate any kind of ball. The inflating skill is described in listing B.10 and this agent's initial state is described in listing B.17;

[4]Keep in mind that the agents shown here are only the agents actually being used to solve the problem. The number of candidate agents (agents that exist in the network but are not actually used in this particular problem) can be quite large.

Consider the ball customisation schema in Figure 3.5 and the listing B.18, which represent a typical problem to be solved in this domain. A client wants a basketball with the whole top area painted green, the top left section with blue stripes, the whole bottom area painted blue and the bottom right section with green stripes.

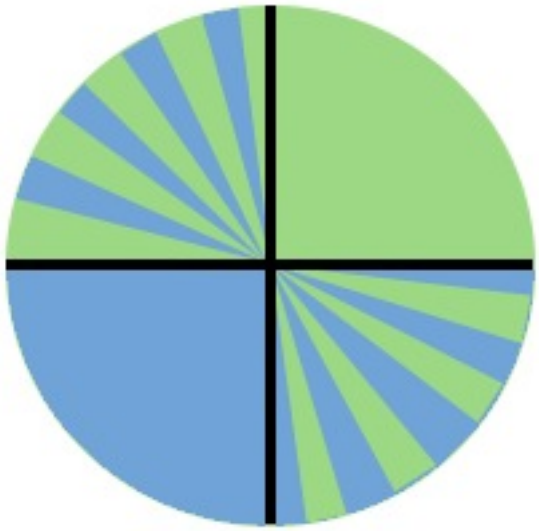

Figure 3.5: *Custom Ball Factory* scenario problem example

In this context, all of the agents described above will have to be used in order to produce the solution plan that will ultimately lead to the production of the customised ball. However, in a typical environment of this scenario, a lot of other agents/skills are available. This can make the process of finding the appropriate skills lengthy and complex.

Let us imagine that the first agent receiving this request is the `infla` agent, responsible for inflating balls. After processing the request, the agent determines that it can contribute to one of the goals, `(inflated ball1)`, by using its action `inflate_ball` and produces the partial solution plan depicted in Figure 3.6.

Each agent in this domain has a particular skill related to the manufacturing of custom sport balls. Besides having a particular skill, each agent has also a pair of actions that guarantees that no two agents will end up working on the same ball at the same time in the final solution plan: `grab_ball` and `drop_ball`.

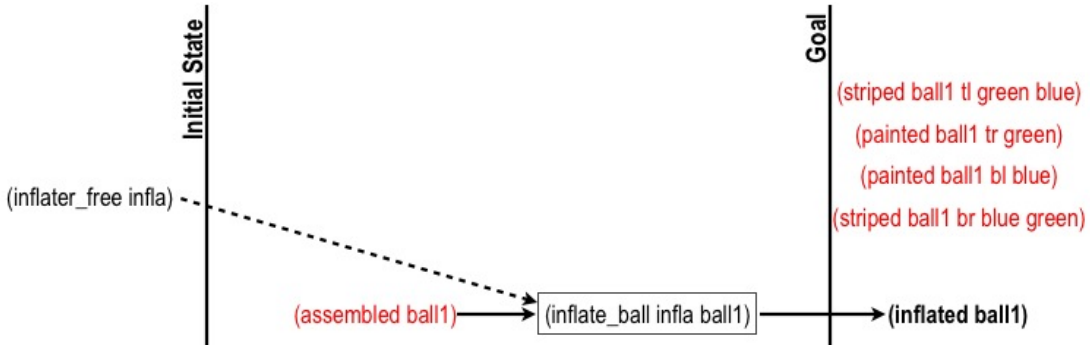


Figure 3.6: Partial solution plan developed by agent `infla`

However, for the sake of simplicity, we removed these auxiliary actions from the following partial plans' diagrams.

The dashed arrows represent propositions that are indirectly connected to the action by the absence of the actions referred above. For example, proposition `(inflater_free infla)` is required by action `grab_ball` that ultimately produces proposition `(inflater_has infla ball1)` that is in fact a precondition for action `inflate_ball`.

At this point, agent `infla` needs to find one or more agents capable of contributing to the open conditions (in red in the figure) of the current solution plan. Considering that the environment in which it is operating can be very large, agent `infla` needs to use an efficient P2P search algorithm to find the appropriate agents that can contribute to the partially-solved problem.

Let us assume it is able to find agent `assemb`, which is capable of contributing to the open condition `(assembled ball1)`. Agent `assemb` will then contribute to the partial solution plan and create the instance described in Figure 3.7.

Agent `assemb` will then use the same discovery process to forward the current partial solution plan to another agent. Since agent `assemb` did not introduce new open conditions, it will try to find agents that can contribute to the remaining unsolved goal propositions. Let us consider that it will send the current solution plan to agent `gs_painter`, which after processing it creates the partial solution plan described in Figure 3.8[5].

[5]Again, to simplify the diagrams, the contributions made by previous agents were removed.

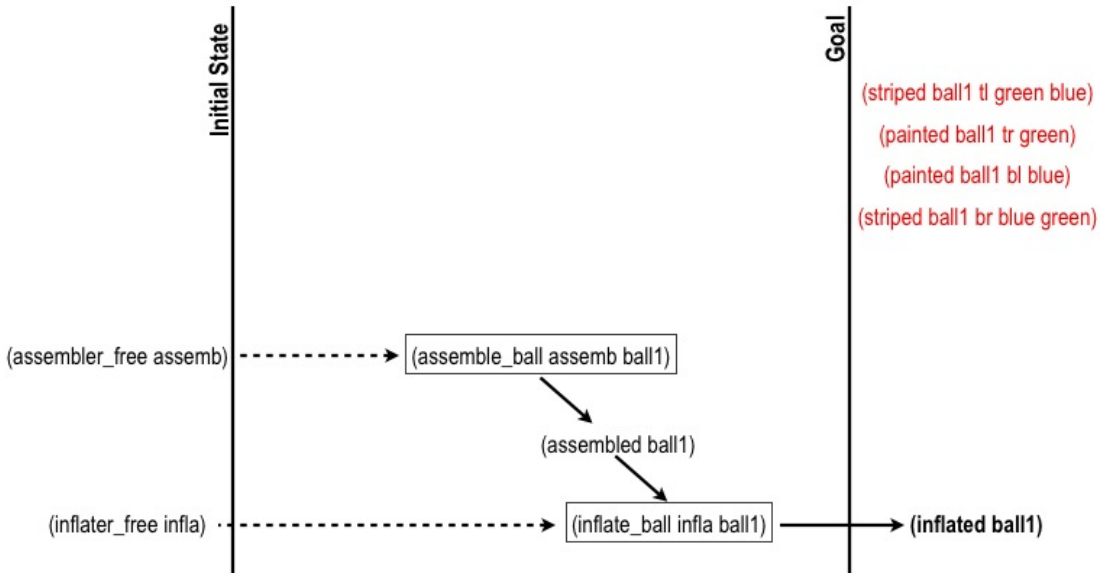


Figure 3.7: Partial solution plan developed by agent `assemb`

The addition of action `paint_stripes` by agent `gs_painter`, which paints the green stripes in the bottom right slice of the ball, introduces a new open condition: `(painted ball1 br blue)`. This open condition reflects the need for the whole bottom section of the ball to be first painted blue, before the stripes are painted. So, agent `gs_painter` needs to find an agent that is able to paint that area of the ball in blue. After searching the network, it is able to find agent `bb_painter`, which after performing the necessary processing of the request, is able to contribute to the current solution plan as depicted in Figure 3.9.

Agent `bb_painter` contributes to the solution plan by adding its action `paint` twice, one for each of the bottom slices of the ball, to paint it blue. After its contribution, only two more goal propositions remain unsolved: `(striped ball1 tl green blue)` and `(painted ball1 tr green)`. These propositions represent the goal of painting the top area of the ball in green and blue stripes on the left section.

These contributions are done in a similar way as shown above for agents `gs_painter` and `bb_painter`, but instead using two other agents with the necessary skills. The action of painting the whole top area of the ball in green is done by agent `trgb_painter` and the blue stripes are painted by agent `bs_painter`. This

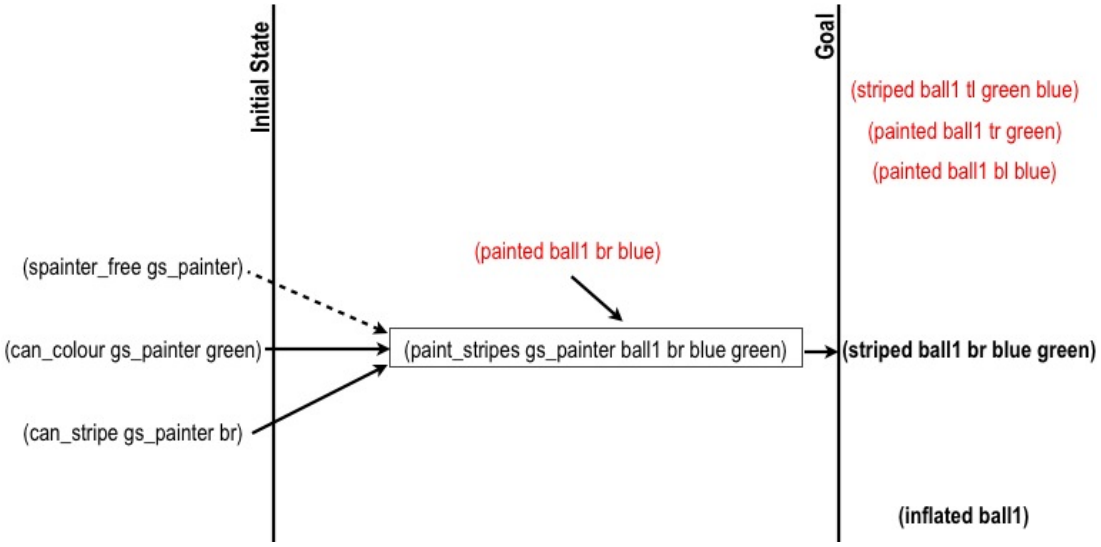


Figure 3.8: Partial solution plan developed by agent `gs_painter`

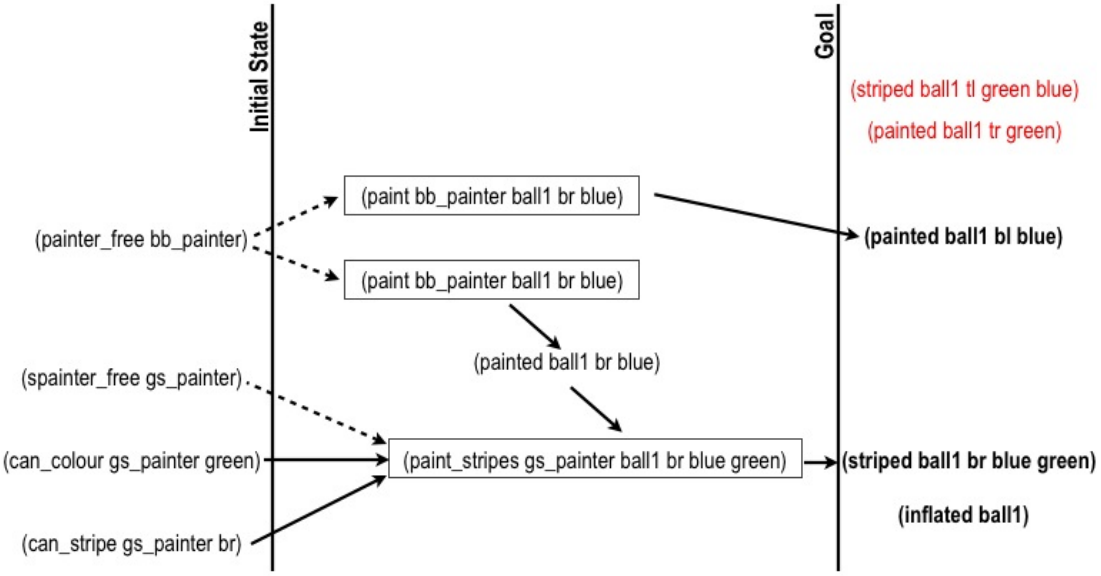


Figure 3.9: Partial solution plan developed by agent `bb_painter`

completes the problem solving process and the final solution plan is presented in listing 3.2.

```
1  <(grab_ball trgb_painter ball1)>
2  <(paint trgb_painter ball1 tr green),(paint trgb_painter ball1 tl
       green)>
3  <(drop_ball trgb_painter ball1)>
4  <(grab_ball bs_painter ball1)>
5  <(paint_stripes bs_painter ball1 tl green blue)>
6  <(drop_ball bs_painter ball1)>
7  <(grab_ball bb_painter ball1)>
8  <(paint bb_painter ball1 br blue),(paint bb_painter ball1 bl green)>
9  <(drop_ball bb_painter ball1)>
10 <(grab_ball gs_painter ball1)>
11 <(paint_stripes gs_painter ball1 br blue green)>
12 <(drop_ball gs_painter ball1)>
13 <(grab_ball assemb ball1)>
14 <(assemble_ball assemb ball1)>
15 <(drop_ball assemb ball1)>
16 <(grab_ball infla ball1)>
17 <(inflate_ball infla ball1)>
18 <(drop_ball infla ball1)>
```

Listing 3.2: Final solution plan in the *Custom Ball Factory* domain example

The sequence of actions presented in this solution plan is only one among many different alternatives. That is, most of these actions can be arranged in a different order and that would not affect the outcome of the solution plan. For example, the order in which the painting process for the top and the bottom of the ball occur is completely irrelevant because these actions do not interfere with each other. However, some conflicts exist and these are avoided by setting a strict order in the solution plan. For example, the inflating action cannot occur before the assembling action and the stripes action cannot occur before the painting action. Hence, the order of these actions in the plan cannot be changed.

## 3.3 Requirements Analysis

Considering both scenarios described in the previous sections, it is clear that the coordination framework inherent to the problem solving process has two main ingredients: a discovery mechanism that allows agents to find other agents that can

help them solve parts of a complex problem; and a partial planning mechanism that allows each agent to determine how it can contribute to parts of that problem and decide which parts have to be solved by other agents with the necessary skills.

It is the combination of these two mechanisms that allows agents to coordinate themselves in a distributed problem solving environment. They can assume, however, different levels of importance for different scenarios. For example, as we have seen in section 3.1, the focus of the *Rescue Agents* scenario is on the planning process due to the high level of cooperation required by the entities of the scenario, whereas in the *Custom Balls Factory* scenario, the focus is on the discovery mechanism because the main task revolves around finding the agents with the necessary skills in a very large network of potential candidates.

In order to make our approach as generic as possible and still guarantee a good performance in many different scenarios, these two mechanisms have to be efficient (rapidly generate responses without imposing a large work load onto the agents), scalable (operate in very large environments) and robust (withstand occasional failures in parts of the system). And more importantly, their combination has to reflect those same properties in the resulting coordination infrastructure.

From the analysis of both scenarios, and especially the *Custom Balls Factory* scenario, the discovery mechanism can be very challenging due to the large size of the environment in which the agents operate. On one hand, a centralised solution is not recommended for its lack of robustness or scalability, as we concluded in section 2.5 of the previous chapter. On the other hand, using a totally distributed solution can help overcome the limitations of a centralised solution but, if not done efficiently, the performance degradation can be high (see section 2.3 of the previous chapter). Thus, it is imperative that the discovery mechanism relies on an efficient algorithm that is able to deal with increasingly large environments.

On scenarios such as the *Rescue Agents* scenario, the focus is on the planning process. In such environments, conflict resolution assumes a very important role and if the agents cannot cooperate efficiently to deliver a fast response to a complex problem, the resulting solution plan may become obsolete very quickly. Moreover, it is also very important that each agent can quickly determine the level of contribution to the problem and not waste much time in the process, since

this may jeopardise the participation of other agents that can actually perform contributions. Thus, the planning mechanism will need to be very efficient in performing the matchmaking process between the agent's actions and the parts of the problem that remain unsolved.

Classical planning approaches that are typically used in one-agent settings, unless carefully adapted to consider large distributed environments, cannot be used (see section 2.1 of the previous chapter). Centralised or organisational-based solutions cannot be considered since, as concluded in section 2.2 of the previous chapter, the approaches that use this kind of strategies compromise the scalability and robustness of the environment.

These requirements, in conjunction with the goals established in Chapter 1 and the conclusions of Chapter 2, are the basis of our research work. The next chapter builds on all these elements and describes our chosen technical approach in detail.

# Chapter 4

# Technical Approach

Efficiently coordinating the distributed problem solving activity of multiple heterogeneous agents in unstructured environments is quite challenging. On one hand, as we have seen in Chapter 2, most approaches rely on some sort of organisational-based model that, even though may generally address the coordination problem, compromises the robustness and scalability of the system. On the other hand, a detailed view of two completely different scenarios, in Chapter 3, has shown that the two main ingredients of our approach, discovery and planning, are equally important to guarantee an efficient domain-independent coordination mechanism. Following these conclusions and the goals established in Chapter 1, we designed and developed a coordination framework on which agents seamlessly rely to solve complex problems[1]. In this chapter, we describe our technical approach for coordinated distributed problem solving in unstructured intelligent agent societies.

The process itself is quite simple and it starts with an agent receiving a request to solve a specific problem, which includes a description of the initial state of the world and the goals to be achieved. This agent processes the request using a planning algorithm to decompose the received problem into sub-problems and identify those sub-problems it can solve with its own skills and those that have to be sent to another agent with other skills. To find another agent to send the non-solved sub-problems, the agent uses a distributed discovery mechanism relying

[1]By complex problems we mean problems that must be solved by more than one agent in cooperation.

on dynamically built and maintained semantic information about the skills of the other agents in the society.

The challenge lies in enabling the agents to perform these tasks without using any structured or centralised elements. To that end, we first propose that agents make use of their idle time to trigger a totally distributed self-organisation process that allows them to learn more about each other's skills and, thus become more prepared to solve problems in the future. It is this self-organisation process, based on the propagation of agents' skills through the network using efficient algorithms, that allows the agents to create and maintain a *semantic overlay network*, that is, a set of links that enable agents to easily find other agents that are semantically related to them. All of these elements, which compose the discovery process of our coordination framework, are described in section 4.1 of this chapter.

In section 4.2, we focus on the other part of our coordination framework, describing the distributed planning process in which agents are able to determine how to partially contribute to complex problems and how to identify the problem parts that need to be forwarded to other agents. It is by using the overlay network described in section 4.1 that this planning algorithm is able to determine which is the best agent to which the partially-solved problem should be forwarded.

Finally, in section 4.3, we substantiate some of decisions that were made through the course of the research work. In particular, we address (in sub-section 4.3.1) the issue of choosing the appropriate network evolution techniques in the self-organisation process described above, the decision regarding the planning algorithm to be used in the distributed planning process (in sub-section 4.3.2) and the decision of which heuristics to use in the backward-search phase of the planning algorithm (in sub-section 4.3.3).

## 4.1 Distributed Agent Discovery

Agents facing problems that can only be solved by the collective effort of different agents in unstructured dynamic networks need an efficient search algorithm to find the agents with the necessary skills or resources to cooperatively solve the problem. This search mechanism is the basis of the discovery process of our coordination framework. With it, agents should not only be able to locate other

necessary agents in the environment but also engage in a self-organisation process that, through information gathering over time, improves the fully distributed agent discovery process itself. However, as networks become larger, search mechanisms become less and less efficient and are unable to adapt to increasingly complex environments.

As an agent enters a network and needs to search for other agents with particular skills or resources, it can employ different kinds of search mechanisms depending on the type of the network in which it is operating. In case of structured networks, the agent must first become acquainted with the agents standing higher in the hierarchy, which are responsible for ensuring the propagation of queries through the entire (or just a part of the) network, and then perform their search requests.

Structured networks have a good search performance but as the network becomes larger, the effort required to constantly adapt the network's structure to account for new agents entering or leaving it, becomes prohibitive. In such cases, this maintenance effort overcomes the gains of good search performance. Moreover, the higher the agent that leaves the network is on the hierarchy, the greater will be the impact on this maintenance process, which in some cases, may be catastrophic.

Unstructured networks do not suffer from this problem since every agent is equal in responsibility and the impact of any of those agents leaving the network is minimal to the rest of the network. However, since there is no network structure, the search mechanisms are often inefficient and/or resource-consuming. In such networks, searching for another agent or resource is often carried out by a *flooding* algorithm (when the agent broadcasts the search query to all of its neighbours) or a *random walk* algorithm (when the agent broadcasts the search query to only a small subset of its neighbours).

Figure 4.1 depicts the comparison between the *flooding* and *random walk* algorithms and helps understand their relative advantages and disadvantages by showing their position in a diagram that explores 3 different dimensions: query response time, network bandwidth and network coverage. Even though the *flooding* technique may retrieve the results faster, it is easy to perceive that it does so at the cost of using the processing power of a large part of the network, whereas

in the *random walk* algorithm, although the impact on the network load is significantly lower, the search latency may increase immensely.

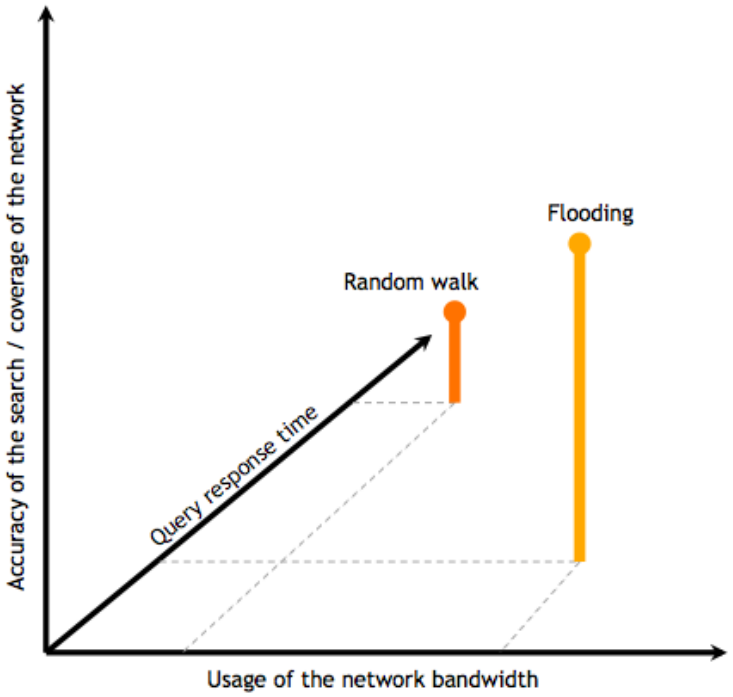


Figure 4.1: Comparison of different search algorithms

Other search mechanisms for unstructured networks (shown in section 2.3), which are based on variations of these two algorithms, explore the trade-off between network load and coverage but the results are more or less similar, that is, as they try to use less of the network bandwidth, they tend to increase the query response time and vice-versa. Our goal is to develop a search algorithm that outperforms any of the currently existing algorithms in, if possible, all 3 dimensions represented in the diagram.

In this section, we present two different search algorithms that we developed aiming to fulfil this goal: the *Priority-based Flooding* (PbF) algorithm (described in sub-section 4.1.1) and the *Iterative Branching Depth-First Search* (IBDFS) algorithm (described in sub-section 4.1.2). We then present, in sub-section 4.1.3, some network evolution techniques and show how these can be used to create a

*semantic overlay network* that can improve the efficiency and accuracy of search queries over time.

### 4.1.1 Priority-based Flooding

This approach is based on the assumption that a *flooding* technique is only inefficient if the network is already overloaded with requests. If the agents in the network are idle, then the *flooding* mechanism is, in fact, the fastest and most complete way of delegating a search query. However, it is difficult for a peer to determine whether or not its neighbours have a heavy workload at a certain moment.

We introduce the concept of *Priority-based Flooding* (PbF), which allows agents to assign a priority to search queries based on their propagation level within the network. The principle of this search mechanism is very simple: agents use the propagation level of a search to calculate the priority of the query, that is, the highest the propagation level, the lower the priority.

Basically, each agent in the network employs the following steps in each iteration of the algorithm (which is formally presented in Algorithm 1):

- The agent determines the priority for each search query currently on the list of search queries to be processed, by using the following formula: $(1/q_{pl})$, with $q_{pl}$ being the query's current propagation level;
- The agent will then process the search query with the highest priority and retrieve the corresponding result;
- If the result of the processing event is `null`, this means that the agent does not hold the answer to the search query and needs to further propagate the request to its neighbours. In that case, after increasing the propagation level of the search query by one unit, the agent forwards the request to all of its neighbours;
- Otherwise, the agent knows the answer and replies to the requesting agent accordingly.

**Algorithm 1** `PBF(Q, N)`: let `Q` be the list of queries currently waiting to be processed (an agent's workload), `N` the list of neighbours, `mq` and `mp` auxiliary variables indicating the maximum priority query and its priority, `p` the priority of a request and `r` the result of a query processing event.

**Require:** $N > 0$
1: $mp \leftarrow 0$
2: $mq \leftarrow 0$
3: **for each** $q_i \in Q$ **do**
4: $p = 1/depth(q_i)$
5: **if** $p > mp$ **then**
6: $mp \leftarrow p$
7: $mq = q_i$
8: **end if**
9: **end for**
10: $r \leftarrow process(mq)$
11: **if** $r = \oslash$ **then**
12: $depth(mq) = depth(mq) + 1$
13: **for each** $n_i \in N$ **do**
14: $reply(forward(mq, n_i))$
15: **end for**
16: **else**
17: $reply(r)$
18: **end if**

In line 4 of alg. 1, we can see that the priority of each request is calculated by using the formula `p = 1/depth(q`$_i$`)`. The method `depth` retrieves the propagation level of a search query, which roughly represents the number of different agents that have already processed the request. Using this technique, agents can efficiently manage their workload by giving priority to local requests (search queries triggered by closer neighbours) in detriment of requests originated by far away agents.

We believe that this is a fair policy, since it relies on the fact that if the propagation level of a search query is high, then the number of agents that have already had access to the search query is also quite high. Hence, it is reasonable to assume that the probability for the search query to have been processed by some other agent with a lower workload is also high.

This approach may allow increasing (or maybe even eliminate the need for) the *Time-To-Live* parameter of *flooding* search queries because requests that have

travelled a lot within the network will simply have such a low priority that the workload of the agents in the network will not be affected by them.

The evaluation of this algorithm is provided in Chapter 5, more particularly, in section 5.1.

### 4.1.2 Iterative Branching Depth-First Search

The *Priority-based Flooding* algorithm focused on reducing the impact of the well-known *flooding* algorithm in the overall network load. The *Iterative Branching Depth-First Search* (IBDFS) explores, instead, the possibility of iteratively increasing the coverage of the network of the opposite approach, the *Depth-First Search* algorithm.

The principle on which this algorithm is based is very simple:

- When initiating a search query, an agent randomly contacts one of its neighbours (this algorithm can also be adapted to consider a `k` number of neighbours, in each iteration). If the neighbour immediately replies with the answer, then the process ends.
- If the neighbour replies stating it has not found the answer, then the agent contacts a second neighbour and so forth.
- Each of the agent's neighbours will employ the exact same process.

This approach (Algorithm 2 depicts the steps of the recursive search mechanism) increases the branching level iteratively on each hop, thus increasing the chances of finding the answer faster, comparatively to the *depth-first search* (DFS) approach. Also, the load effect of this algorithm on the network is much lower than a *flooding*-based technique. However, by delaying the propagation of the search query this algorithm may not retrieve answers as fast as the *flooding* algorithm.

In order to easily understand the difference between the two algorithms, Figure 4.2 presents a comparison of the number of agents reached (darker nodes) in a search query with a hop count of 3, that is, when the algorithms have made 3 iterations.

**Algorithm 2** `IBDFS(q, N)`: let `q` be the query to be processed, `N` the list of neighbours and `r` the result of a query processing event.

**Require:** $N > 0$

1: $r \leftarrow \oslash$
2: **if** $\neg processed(q)$ **then**
3: $\quad r \leftarrow process(q)$
4: $\quad reply(r)$
5: $\quad processed(q) \rightarrow true$
6: **end if**
7: **if** $r = \oslash$ **then**
8: $\quad randomly\ select\ n_i\ \in\ N$
9: $\quad r_{n_i} \leftarrow forward(q, n_i)$
10: $\quad$ **if** $r_{n_i} = \oslash$ **then**
11: $\quad\quad IBDFS(q, N - n_i)$
12: $\quad$ **end if**
13: **end if**

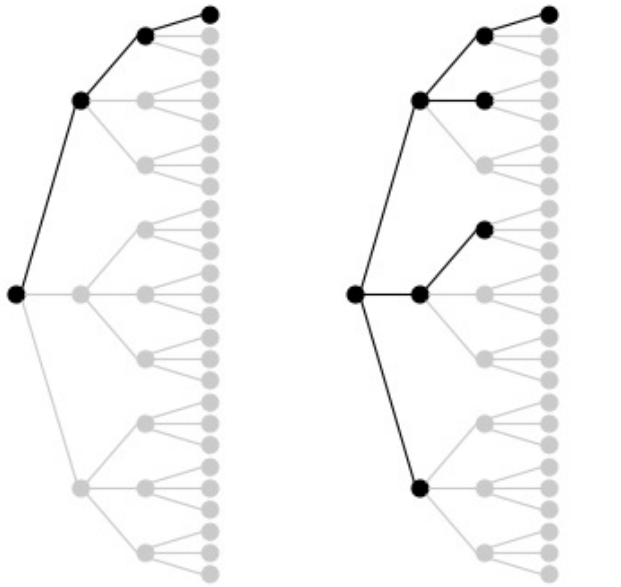


Figure 4.2: Comparison between the Depth-First Search algorithm and the *Iterative Branching Depth-First Search* algorithm with a hop count of 3, *i.e.*, when the algorithms have made 3 iterations.

A detailed evaluation of this algorithm is provided in Chapter 5, more particularly, in section 5.1.

### 4.1.3 Network Evolution and Semantic Overlay Networks

In the beginning of the discovery process, agents in a peer-to-peer (P2P) network do not have enough information about other agents. As they go along, the interactions between them become valuable sources of information that can be used to improve the performance of future searches. Furthermore, the use of informed search techniques scales a lot better throughout time as agents develop connections with other agents discovered in previous interactions (Fletcher & Sheth, 2004).

To improve the performance of the proposed search mechanisms, we also propose some adaptation procedures that we believe will improve the searches throughout time. These procedures contribute to the evolution of the network by triggering a self-organisation process that will improve future searches (thus reducing the query response time and the network bandwidth usage and maximising the accuracy of the results), ultimately leading to the dynamic creation of a *semantic overlay network*.

A *Semantic Overlay Network* (Crespo & Garcia-Molina, 2005) is an abstract layer that represents a set of semantic relationships established between agents that are randomly connected at a lower level of a peer-to-peer network. Basically, a *semantic overlay network* represents the interconnections of semantically related network nodes. Figure 4.3 shows an example of a *semantic overlay network* that was created on top of a peer-to-peer network. A semantic link (green lines in figure 4.3) between two agents represents a connection that denotes that one agent is related to the other in terms of similarity or dependency of their skills or resources.

The idea is to create an adaptable search algorithm that improves its performance over time ultimately leading to the creation of a complete *semantic overlay network*, as depicted in Figure 4.4, which revisits the 3-dimensional analysis of figure 4.1.

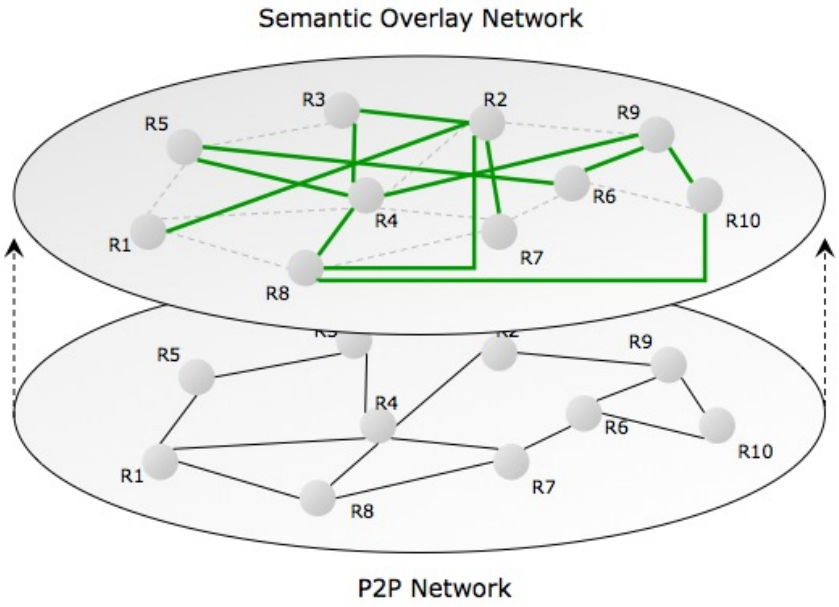


Figure 4.3: A *semantic overlay network* as a layer of the physical P2P network.

In order for an agent to improve its participation in future searches, it is important that it caches previous search contributions. For example, as a query response travels back to the requester agent, all agents in that specific path can either store the response themselves or cache a link to the agent which has the response, thus working as a referral for future searches.

However, after some time contributing to search queries, it may happen that agents hold a huge cache of referrals that becomes intractable as they contribute more and more throughout time. In order to avoid loss in performance due to the size of the cache, agents can store only a fixed number of references and decide which ones to store based on a metric, such as the frequency of the request. Even though rare requests have lower performance in this process, frequent search queries will be optimised, which globally seems to be a good option.

An alternative approach can be based on a direct link between the responder agent and the requester agent. If we consider, for example, the *priority-based flooding* algorithm, we see that this causes a massive generation of reply messages (line 14 of alg. 1). To avoid this situation, the search mechanism can be changed so that the query response is returned directly to the query requester, instead of being carried back through the original path. For example, if agent $\alpha$ has the

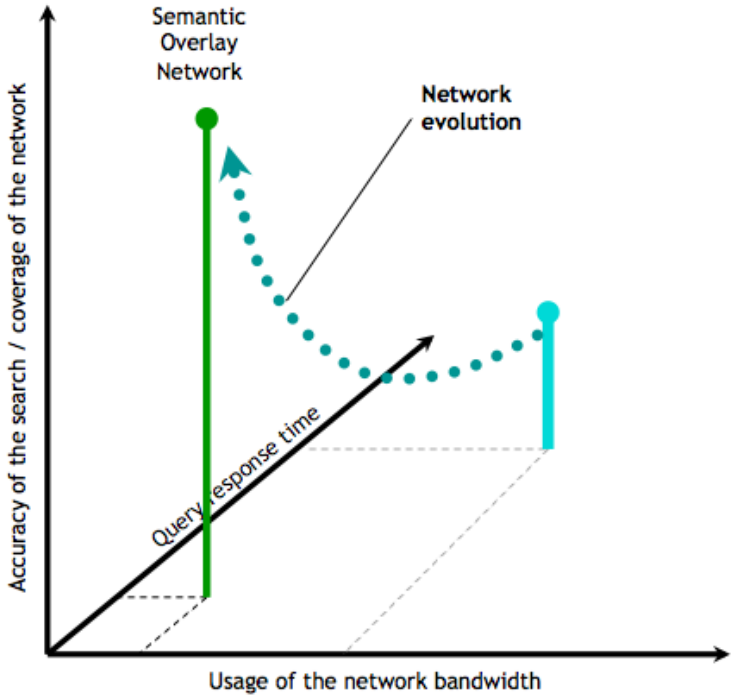


Figure 4.4: Network evolution techniques as a *semantic overlay network* generation process.

response for the query made by agent $\beta$, $\alpha$ will directly send the response to $\beta$. Even though the agents on the original request path will not learn the result of the query, the result will reach the requester agent faster and a lot of messages can be saved.

Furthermore, the agents that participated in the search, even if just for forwarding or propagating the request, can assume that, after some time, the requester agent has already received the necessary response. Hence, future similar searches (for example, agent $\gamma$ requesting the same contents as $\beta$) can be referred to the previous requester agent ($\beta$), which in turn can refer it to the original responder agent ($\alpha$) or provide itself the response directly (to $\gamma$).

These techniques, which are analysed and evaluated in section 4.3.1, can be very useful for improving the agent discovery process over time. But not only do they improve the search mechanism, they also allow agents to collect valuable information about the agents/skills/resources existing in the network at any given

time. Considering that, in distributed problem solving environments, agents need to collaborate, it is very important that they can maintain a record of where the required skills are. The use of semantic links is just the ideal solution for that problem.

In our approach, we use a *semantic overlay network* to establish relationships between agents to facilitate their discovery. These relationships reflect semantic dependencies between agent skills. Agent skills are actions that can be executed in the environment in which they operate. An action has inputs and outputs and can only be executed if its preconditions are met. After execution, an action produces some effects (propositions guaranteed to be true after execution) in the environment. These properties, which describe actions in a meaningful way, are used to optimise the discovery process in collaborative environments.

As agents enter a network, they engage in a self-organisation process in order to build and maintain a *semantic overlay network* that will ultimately establish all the dependency relationships amongst agents (and their actions). The self-organisation process is triggered by the agents as they propagate information regarding their own skills (using one of the network algorithms described above).

During the self-organisation process, each agent uses a simple rule to determine whether or not other agent skills should be semantically linked to its own skills. Agent $\alpha$'s action $a$ should be semantically linked to agent $\beta$'s action $b$ if $b$'s effects (denoted as $effects(b)$) contribute to achieve $a$'s preconditions (denoted as $precond(a)$), as illustrated by the following expression[2]:

$$\exists\ c\ [(c \in precond(a) \wedge (effects(b) \vdash c)] \tag{4.1}$$

The main purpose of this process is to allow a network of otherwise unrelated agents to self-organise, such that each agent knows exactly where the actions on which its own actions depend (or contribute to) are. Also, during the self-organisation process, it is natural that each agent will also acquire knowledge not only related to its own actions but to the ones of other agents.

[2]We consider that preconditions and effects are sets of propositions that represent their conjunction

## 4.2 Distributed Planning

The self-organisation process described in the previous section creates links between agent actions, thus dynamically building a *semantic overlay network*. This abstract layer is the basis of the discovery process of the proposed approach to coordinated distributed problem solving, in large-scale unstructured environments. It helps agents find each other relying on semantic relationships between them.

However, for an agent to solve a specific problem, it needs an algorithm that is capable of dealing with partial knowledge of the domain (that is, restricted to the agent's local knowledge) and to derive its contributions to the referred problem. Moreover, that algorithm should also take into account that the agent is working together with other agents in order to solve a problem that can only be solved (or solved more quickly or efficiently) through the collective effort of several agents. This means the algorithm must also consider other agents' contributions and resolve any conflicts that may arise in the distributed problem solving process.

The proposed approach to coordinated distributed problem solving uses a planning algorithm to decompose a problem into sub-problems, to identify those sub-problems that can be solved by the agent, to identify those that must be delegated to other agents, and coordinate the whole cooperative activity avoiding conflicts. With the purpose of selecting the best alternative, we have tested several different planning algorithms in a distributed setting. From that analysis (which is presented in section 4.3.2) we have concluded that the *Graphplan* algorithm is the most efficient approach.

This section describes the *Graphplan* algorithm and the extensions we have made to it. First of all we describe, in sub-section 4.2.1, a very simple example of a planning problem from the blocks world. We use such a simple example to avoid producing illegible diagrams, difficult if not impossible to grasp. However, the real problems in which we have tested our approach are far more complex. We then describe, in sub-section 4.2.2, the fundamental aspects of planning graphs and the *Graphplan* algorithm. In sub-section 4.2.3, we present our distributed version of the *Graphplan* algorithm. And finally, in sub-section 4.2.4, we present an alternative version of the distributed *Graphplan* algorithm, which uses *means-ends* analysis.

### 4.2.1 A Blocks World planning problem example

The Blocks World is one of the most famous planning domains in artificial intelligence and has often been used to describe the behaviour of some planners in particular examples. Imagine a set of blocks (or cubes) sitting on a table. The goal is to build some desired structure by vertically stacking blocks using a mechanical hand. The catch is that only one block may be moved at a time: it may either be placed on the table or placed on top of another block. Any blocks that are, at a given time, under another block cannot be moved.

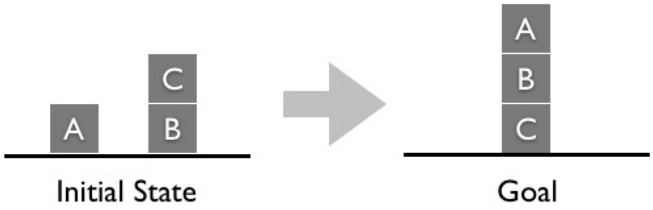


Figure 4.5: A planning problem example in the Blocks World domain.

Consider the following example in this domain, which is depicted in Figure 4.5. In the initial state, blocks $A$ and $B$ are placed on the table and block $C$ is placed on top of $B$; the goal is to reach a state in which $C$ is placed on the table, $B$ is on top of $C$ and $A$ is on top of $B$.

In this domain, the following predicates are used to represent states of the world[3]:

- `(clear ?b)` – indicates that there is no block on top of block `?b`;
- `(on ?a ?b)` – indicates that block `?a` is on top of block `?b`;
- `(ontable ?b)` – indicates that block `?b` is placed on the table;

Considering the predicates shown above, we now describe, in Listing 4.1, the problem shown in figure 4.5 in PDDL:

[3]All listings presented here are written in PDDL.

```
1 (define (problem blocks1) (:domain Blocks)
2   (:objects a b c )
3   (:init (ontable a) (clear a) (ontable b) (on c b) (clear c) )
4   (:goal (and (ontable c) (on b c) (on a b) ) ) )
```

Listing 4.1: Description of the Blocks World example problem in PDDL

In order to create a plan that, starting from the initial state, achieves the desired goal, we need to have actions that produce certain effects on the world. In this example, we will consider the following actions (a detailed description of these actions in PDDL is provided in listing 4.2 below):

- `(move ?a ?b ?c)` – this action is used to make the mechanical hand move block `?a` from the top of block `?b` onto block `?c`;
- `(stack ?a ?b)` – this action is used to make the mechanical hand move block `?a` from the table onto block `?b`;
- `(unstack ?a ?b)` – this action is used to make the mechanical hand move block `?a` from the top of block `?b` onto the table.

```
1 (define (domain Blocks)
2 (:action move
3   :parameters (?a ?b ?c)
4   :precondition (and (on ?a ?b) (clear ?c) (clear ?a))
5   :effect (and (clear ?b) (on ?a ?c)
6       (not (on ?a ?b)) (not (clear ?c)) ) )
7 (:action stack
8   :parameters (?a ?b)
9   :precondition (and (clear ?b) (ontable ?a) (clear ?a))
10  :effect (and (on ?a ?b)
11      (not (clear ?b)) (not (ontable ?a)) ) )
12 (:action unstack
13  :parameters (?a ?b)
14  :precondition (and (clear ?a) (on ?a ?b))
15  :effect (and (clear ?b) (ontable ?a)
16      (not (on ?a ?b)) ) ) )
```

Listing 4.2: Description of the Blocks World example domain in PDDL

This will be the example used throughout the rest of this chapter to clearly describe the behaviour of the different algorithms that are presented here.

### 4.2.2 Planning Graphs and the *Graphplan* algorithm

Given an initial state, a set of goal propositions[4] and a set of available actions, a *planning graph* (Blum & Furst, 1997) consists of a directed, levelled graph where levels alternate between proposition levels containing proposition nodes and action levels containing action nodes, as such: $\langle P_0, A_1, P_1, \ldots, A_i, P_i \rangle$.

The first level ($P_0$) is a proposition level composed of proposition nodes corresponding to the initial state. The second level ($A_1$) is an action level composed of action nodes, one for each action whose preconditions are satisfied by the propositions in the first level ($P_0$). The third level ($P_1$) is a proposition level composed of proposition nodes that represent the propositions resulting of the effects of the actions in the second level.

Since the actions are only connected to the propositions they change, the persistence of propositions during action execution is not explicit in the planning graph. To explicitly make propositions persist from one proposition level to the next one, at each level $P_i$, each proposition $p \in P_i$ is propagated to the next level $P_{i+1}$ by a dummy action `no-op` that has a single precondition and a single positive effect $p$.

Figure 4.6 shows the state of the *planning graph* in the Blocks World example, after the first level expansion. Black lines coming from propositions to actions (squares in the figure) represent the action preconditions, and black lines coming from actions to propositions represent action positive effects. Dashed lines represent action negative effects. `No-op` actions are not represented in the figure for the sake of clarity. Red lines in the figure, represent *mutexes*, which are explained below.

As we can see in the figure, only actions `(stack a c)`, `(move c b a)` and `(unstack c b)` can occur at the first action level of the *planning graph* ($A_1$) since only these have their preconditions satisfied in the first proposition level

[4]We use the STRIPS (Fikes & Nilsson, 1971) representation under the assumption of a deterministic and fully observable domain.

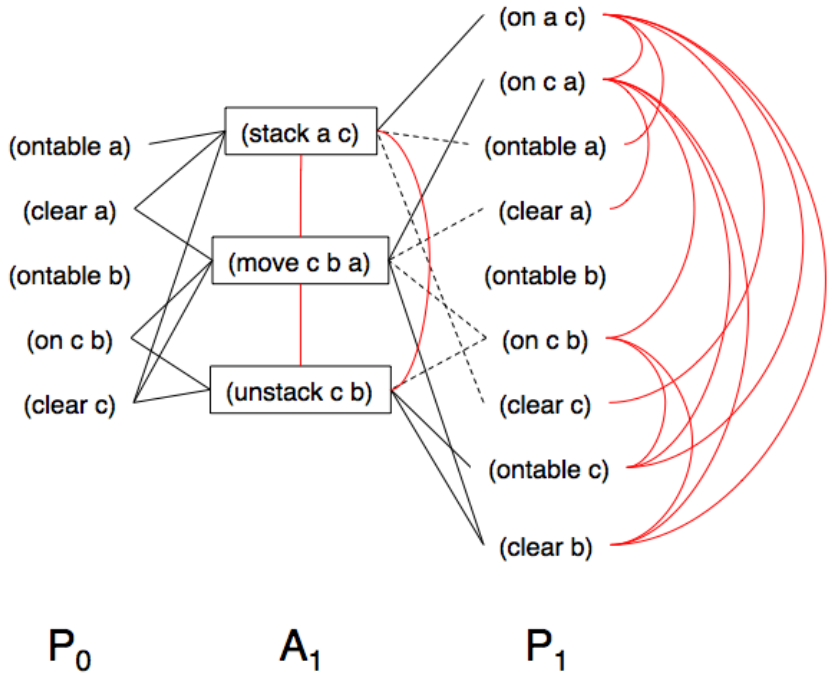


Figure 4.6: First level expansion of the *planning graph* in the Blocks World example problem.

($P_0$). These actions will then add their effects onto the second proposition level ($P_1$), namely, `(on a c)` for action `(stack a c)`, `(on c a)` and `(clear b)` for action `(move c b a)` and `(ontable c)` and `(clear b)` for action `(unstack c b)`. The propositions that are in level $P_0$ are also added to level $P_1$ due to the `no-op` actions (which are not represented in the figure).

The *planning graph* is built this way until a proposition level is reached that includes all propositions of the goal state. A *planning graph* does not represent a valid plan for a planning problem. Instead, it uses the principles of *independence* and *mutual exclusion* – or *mutex* – to drastically reduce the search space and help finding a valid plan faster.

We can say that two actions $a$ and $b$ are independent if and only if the following

two conditions are met[5]:

$$effects^{-}(a) \cap [precond(b) \cup effects^{+}(b)] = \oslash \quad (4.2)$$

$$effects^{-}(b) \cap [precond(a) \cup effects^{+}(a)] = \oslash \quad (4.3)$$

Two actions `a` and `b` in level $A_i$ are *mutex* if either `a` or `b` is dependent of the other or if a precondition of `a` is *mutex* with a precondition of `b`. Two propositions `p` and `q` in $P_i$ are *mutex* if every action in level $A_{i-1}$ that has `p` as a positive effect (including `no-op` actions) is *mutex* with every action that produces `q`. Basically, a *mutex* represents two elements that cannot occur at the same level of a valid plan at the same time. The set of *mutex* relations at a proposition level $P_i$ and action level $A_i$ are denoted respectively $\mu P_i$ and $\mu A_i$.

It can be seen in figure 4.6 that all pairs of actions in level $A_1$ are *mutex*, that is, are mutually exclusive, because each of them deletes at least one precondition of the other. For example, action `(move c b a)` deletes the proposition `(clear a)`, which is a precondition of `(stack a c)`. In level $P_1$, several *mutexes* occur between propositions. For example, proposition `(on a c)` is *mutex* with proposition `(on c a)` because the only actions that have these propositions as positive effects are actions `(stack a c)` and `(move c b a)`, which in turn are *mutex*.

*Graphplan* (Blum & Furst, 1997) is an example of the use of a *planning graph*. The *Graphplan* algorithm iteratively expands the *planning graph* by one level[6] and then searches backward from the last level of this graph for a solution. The search procedure looks for a set of non-*mutex* actions that achieve the goal propositions. Preconditions of the chosen actions become the new goal propositions and the process continues. A failure to meet the goal at some level $i$ leads to backtrack over all other subsets of actions in level $i + 1$. If the first level is successfully reached, then the corresponding action sequence is a solution plan.

Figure 4.7 represents the final *planning graph* for the Blocks World example. The elements in light grey represent all reachable actions/propositions starting

[5]We denote $precond(a)$ as the preconditions of an action $a$, and respectively $effects^{+}(a)$ and $effects^{-}(a)$ as the positive and negative effects of $a$.

[6]With the exception of the first expansion, which is done until a proposition level is reached where all goal propositions are included and no pairs of them are *mutex* since it does not make sense to start searching for a plan in a graph that does not reach the goal state.

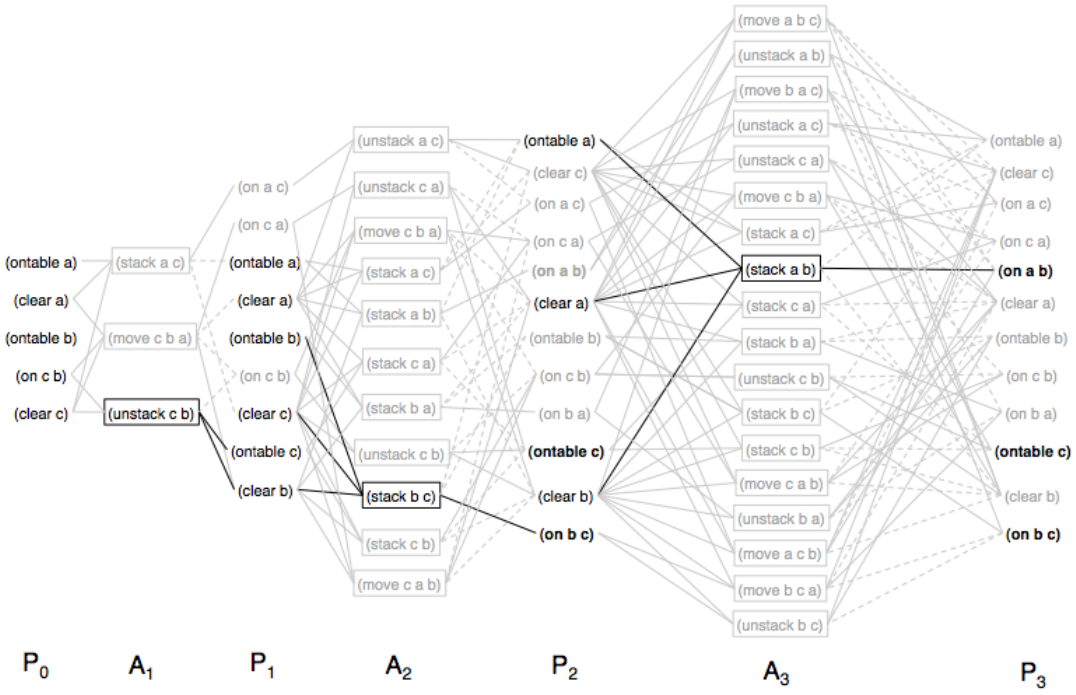


Figure 4.7: Final *planning graph* in the Blocks World example problem.

from the initial state and generated by the expansion process. The expansion process terminates at level $P_3$ because all of the goal propositions occur at this level and no pair of them is *mutex*. Black elements represent the actions/propositions used in the backward search process, which ultimately finds the final solution plan (shown in Listing 4.3).

```
1 <(unstack c b)>
2 <(stack b c)>
3 <(stack a b)>
```

Listing 4.3: Final solution plan of the Blocks World example problem.

The iterative graph expansion and the search processes are pursued until either a plan is found or the search reveals that no solution can be found in the *planning graph*. The algorithm knows that there is no solution to a given problem when the *planning graph* levels-off, that is, when it reaches a fixed-point level `k` that is the smallest `k` such that $|P_{k-1}| = |P_k|$ and $|\mu P_{k-1}| = |\mu P_k|$.

*Graphplan* has revolutionised automated planning research mainly because of its simple, elegant algorithm and its representation of planning problems that created the basis for an extremely fast planner (Weld, 1999). Nevertheless, the algorithm applies to the one-agent planning paradigm and does not explore the potential of being used in distributed settings.

### 4.2.3 Distributed Graphplan

In the centralised version of the *Graphplan* algorithm, the planning agent has full knowledge of the available actions. However, in a distributed environment, each agent only has knowledge of its own actions. So, we modified the algorithm to take into account partial contributions to the development of the *planning graph*. The process is carried out as follows (Algorithm 3 details the expansion process engaged by each agent[7]):

- An agent receiving a problem solving request, which includes a description of the initial state and a set of goals, creates the first proposition level ($P_0$ - line 2 of alg. 3) that is composed of all propositions of the initial state (this is only done by the first agent that receives this request);

- The agent then determines which of its own actions can be added to each action level $A_i$ (line 5 of alg. 3) and corresponding propositions to level $P_i$ (line 6 of alg. 3) of the *planning graph*;

- *Mutexes* are calculated for all possible pairs of added actions and of those with the actions in level $A_i$. The *mutexes* between actions already present in level $A_i$ do note have to be recalculated. An identical process is carried out for propositions (see lines 7 and 9 of Algorithm 3 for details).

- When the agent is unable to make further contributions to the *planning graph* (i.e., when the *planning graph levels-off* – line 11 of the alg.), it analyses the open propositions (to which it was unable to contribute) and

[7]Sets in the algorithm with a superscripted 2 in the name, represent all possible pairs of the elements of those sets.

**Algorithm 3** `Expand(i, PG)`: Let `i` be the current level of expansion in the planning graph, `I` the set of propositions in the initial state, `G` the set of goal propositions, `PG` a *planning graph* with the structure $\langle P_0, A_1, \mu A_1, P_1, \mu P_1 \ldots, A_n, \mu A_n, P_n, \mu P_n \rangle$ and `A` the set of actions the agent knows:

1: **if** $i = 0$ **then**
2: $PG\langle P_0 \rangle \leftarrow I$
3: `Expand`$(1, PG)$
4: **else**
5: $A' \leftarrow \{a \in A \mid precond(a) \subseteq P_{i-1}$ and $precond_2(a) \cap \mu P_{i-1} = \oslash\}$
6: $P' \leftarrow \{p \mid \exists\ a \in A' : p \in effects^+(a)\}$
7: $\mu A_i \leftarrow \{(a,b) \in A'^2$ and $(a \in A', b \in A_i)$, $a \neq b \mid effects^-(a) \cap [precond(b) \cup effects^+(b)] \neq \oslash$ or $effects^-(b) \cap [precond(a) \cup effects^+(a)]$ or $\exists\ (p,q) \in \mu P_{i-1} : p \in precond(a), q \in precond(b)\}$
8: $A_i \leftarrow A_i \cup A'$
9: $\mu P_i \leftarrow \{(p,q) \in P'^2$ and $(p \in P', q \in P_i) \mid \forall\ a, b \in A_i, a \neq b : p \in effects^+(a), q \in effects^+(b) \Rightarrow (a,b) \in \mu A_i\}$
10: $P_i \leftarrow P_i \cup P'$
11: **if** $|P_{i-1}| = |P_i|$ and $|\mu P_{i-1}| = |\mu P_i|$ **then**
12: `AnalyseAndForward`$(PG)$
13: **else**
14: **if** $(\forall g \in G) \mid g \in P_i$ and $G^2 \cap \mu P_i = \oslash$ **then**
15: *return PG*
16: **else**
17: `Expand`$(i+1, PG)$
18: **end if**
19: **end if**
20: **end if**

forwards the partial *planning graph* to an agent chosen from a set of *appropriate* agents (obtained using the agent discovery mechanism supported by the *semantic overlay network* – line 12 of alg. 3);

- The new agent receiving the *planning graph* will execute these same steps up to a point where a level $P_i$ in the graph is reached where all goal propositions exist and none of which is *mutex* with any other (line 14 of alg 3), or until a certain terminating condition holds.

The `AnalyseAndForward` procedure in the algorithm (line 12) encapsulates the choice of the agent to which the partially-filled planning graph should be sent. There are several different ways as to how this process can be carried out and we provide a detailed analysis on this in Chapter 5, more particularly, in sub-section 5.2.3.

The termination of this overall expanding process in a distributed environment is not trivial. In the centralised version an agent can declare that a problem is impossible, if the graph *levels-off*. For an agent with only partial knowledge of the world, it is impossible to know if a *levelled-off* graph means that the problem is impossible or if it simply means that the agent does not have the necessary skills to complete it.

This could lead to an indefinite process of forwarding partially solved problems between agents. To avoid this situation, we use a similar mechanism as the one used in P2P search algorithms, where a *time-to-live* (TTL) parameter is used to specify the allowed number of times the request may be forwarded without it being updated with new contributions. Once that TTL parameter expires, the problem is considered impossible and the requester agent is duly informed.

Once a *planning graph* reaches a point where all goal propositions exist and none of which are *mutex*, it is up to the agent holding the *planning graph* at that time to execute the backward search (starting from the goal propositions) that will find a valid solution plan. The agent can also request the assistance of other agents in the backward search. In such cases, each agent will use a different heuristic in the process (an analysis of the heuristics used in the backward search is presented in sub-section 4.3.3). Algorithms 4 and 5 carry out this whole process.

Algorithm 4 takes as input a planning graph, a current set of goal propositions and a current level index. It extracts a set of actions that achieves the goal propositions by recursively calling Algorithm 5 (line 7). If it succeeds in reaching level 0, then it returns an empty sequence (lines 1 and 2), from which pending recursions successfully return a solution plan.

The *mutex* relation between propositions provides only forbidden pairs, not tuples. But it might be the case that the search process shows that a tuple of more than two propositions corresponding to an intermediate sub-goal fails. To avoid analysing the same (invalid) tuple more than once, which might occur due

**Algorithm 4** `Extract(PG, G, i)`: Let `PG` be a *planning graph* with the structure $\langle P_0, A_1, \mu A_1, P_1, \mu P_1 \ldots, A_n, \mu A_n, P_n, \mu P_n \rangle$, `G` the current set of goal propositions, `i` the current level being analysed and $\pi_i$ a set of actions that achieve propositions of `G`:

```
1: if i = 0 then
2:    return ⟨⟩
3: else
4:    if G ∈ ∇(i) then
5:       return ⊘
6:    else
7:       π_i ← SearchGP(PG, G, ⊘, i)
8:       if π_i ≠ ⊘ then
9:          return π_i
10:      else
11:         ∇(i) ← ∇(i) ∪ G
12:         return ⊘
13:      end if
14:   end if
15: end if
```

to the backtracking and the iterative deepening of the backward search process, alg. 4 records any information regarding failed tuples (in the hashtable denoted by $\nabla$ – in line 11) and checks each current goal with respect to these recorded tuples (in line 4) to save time in future searches.

Algorithm 5 selects each goal proposition `p` at a time (line 9) and from the `resolvers` of `p`, that is, actions that achieve `p` and that are not *mutex* with actions already selected for that level, it chooses one action `a` (line 14) that tentatively extends the current subset $\pi_i$ through a recursive call at the same level (line 15). This is performed on a subset of goals minus `p` and minus all positive effects of `a` in `g`. If a failure regarding this choice occurs, a backtrack over other alternatives for achieving `p` (if any) or a backtrack further up (if all `resolvers` of `p` have been tried) is performed. When `g` is empty (line 1), then $\pi_i$ is complete. At this point, the search recursively tries to extract a solution for the following level `i-1` (line 2). This process carries on until the first proposition level is reached successfully and a final solution plan is extracted from the planning graph.

In order to fully understand the specifics of the distributed approach, let us recall the Blocks World example. Obviously, we will not consider a centralised

**Algorithm 5** `SearchPG(PG, g, `$\pi_i$`, i)`: Let `PG` be a *planning graph* with the structure $\langle P_0, A_1, \mu A_1, P_1, \mu P_1 \ldots, A_n, \mu A_n, P_n, \mu P_n \rangle$, `G` the current set of goal propositions, $\pi_i$ a set of actions that achieve propositions of `G`, `i` the current level being analysed, $A_i$ the action level `i`, $\mu A_i$ the *mutexes* between actions in $A_i$ and $\Pi$ the current solution plan:

1: **if** $G = \oslash$ **then**
2: $\Pi \leftarrow$ `Extract`$(PG, \cup\{precond(a) \mid \forall\ a \in \pi_i\}, i-1)$
3: **if** $\Pi = \oslash$ **then**
4: *return* $\oslash$
5: **else**
6: *return* $\Pi.\langle \pi_i \rangle$
7: **end if**
8: **else**
9: select any $g \in G$
10: $resolvers \leftarrow \{a \in A_i \mid g \in effects^+(a)\ and \ \forall\ b \in \pi_i : (a, b) \notin \mu A_i\}$
11: **if** $resolvers = \oslash$ **then**
12: *return* $\oslash$
13: **else**
14: select any $a \in resolvers$
15: *return* `SearchGP`$(PG, G - effects^+(a), \pi_i \cup a, i)$
16: **end if**
17: **end if**

approach in which one agent is aware of all the actions existing in the domain. Instead, the 3 actions are distributed through 3 different agents: agent *mover* is the owner of action `(move ?a ?b ?c)`; agent *stacker* is the owner of action `(stack ?a ?b)`; and agent *unstacker* is the owner of action `(unstack ?a ?b)`. We also assume that the agents have already triggered the self-organisation process that allowed them to create the *semantic overlay network* depicted in Figure 4.8.

The figure represents all semantic relationships between actions and propositions in the Blocks World domain. Actions (represented by round-cornered grey boxes) and propositions are represented in the *semantic overlay network* by their name and arity (number of parameters they have). Also, each action has a list of the agents that are able to perform it. Arrows coming from propositions to actions represent preconditions, and arrows coming from actions to propositions represent positive effects. Negative effects are represented by dashed arrows. Using the information present in this *semantic overlay network*, the agents can automatically determine which agent should be contacted to satisfy a specific

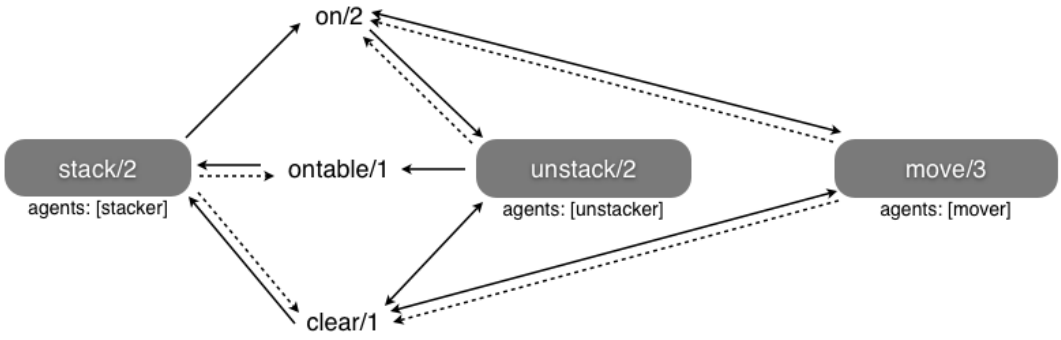


Figure 4.8: Semantic Overlay Network in the Blocks World example problem.

condition in a partially-solved plan.

Considering the example in the Blocks World domain, let us imagine that the first agent to receive the request to solve the problem is agent *mover*. Using alg. 3, the agent produces the partial *planning graph* described in Figure 4.9.

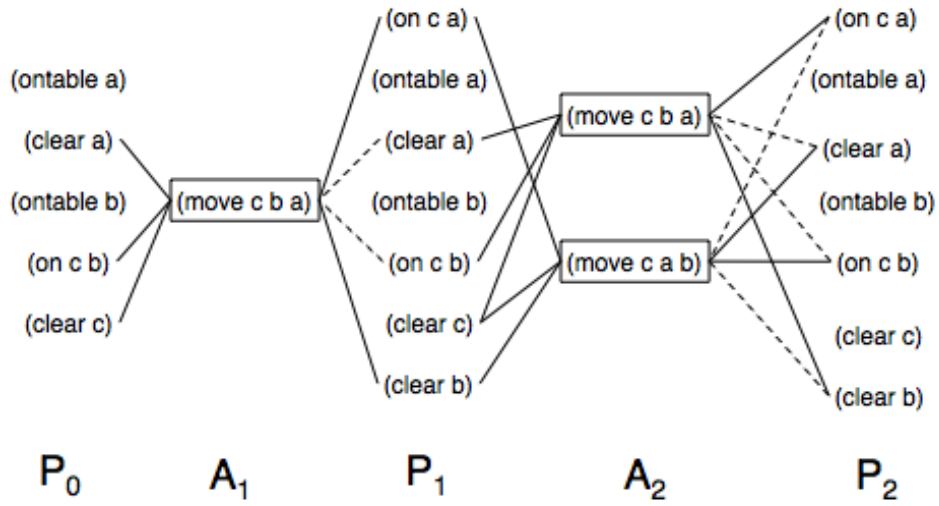


Figure 4.9: Agent *mover*'s contribution to the Blocks World example problem.

Agent *mover*, being able to perform action `move/3`, contributes to the *planning graph* by adding instances of its action to the first and second levels. The agent stops the expansion process at level $P_2$, because the *planning graph* has *levelled-*

*off*, *i.e.*, $|P_1| = |P_2|$ and $|\mu P_1| = |\mu P_2|$.

Since agent *mover* is unable to perform any more contributions at this point, it analyses the current *planning graph* and, using the information in the *semantic overlay network*, determines the agent that should be contacted next. Both actions `unstack/2` (owned by agent *unstacker*) and `stack/2` (owned by agent *stacker*) can be added to the graph in the first action level ($A_1$). Agent *mover* could use an heuristic to determine which one of the agents should the partially-filled graph be sent to. However, at this point, we assume it simply uses a non-deterministic approach and randomly selects one of the alternatives.

Let us assume the partially-solved problem is sent to agent *unstacker*. Using alg. 3, the agent determines it can contribute to the problem using its own action `unstack/2`, as depicted by Figure 4.10.

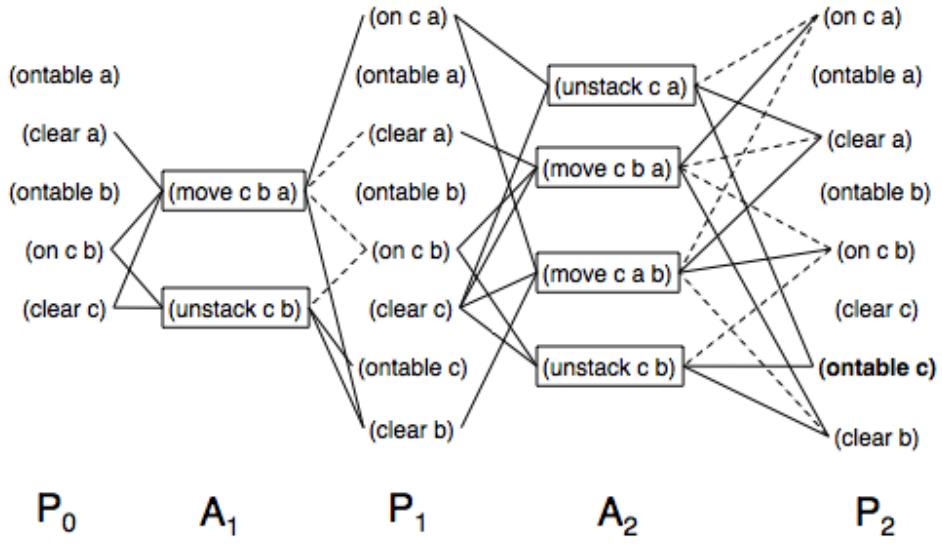


Figure 4.10: Agent *unstacker*'s contribution to the Blocks World example problem.

As can be seen in the figure, the *planning graph levels-off* again at level $P_2$. However, one of the goal propositions, `(ontable c)`, has now been achieved. Since the graph is not yet complete, as some of the goal propositions remain to be satisfied, this agent has to determine which agent this partially-filled graph should be sent to. At this point, the *planning graph* is sent to agent *stacker*,

which produces the *planning graph* described in Figure 4.11.

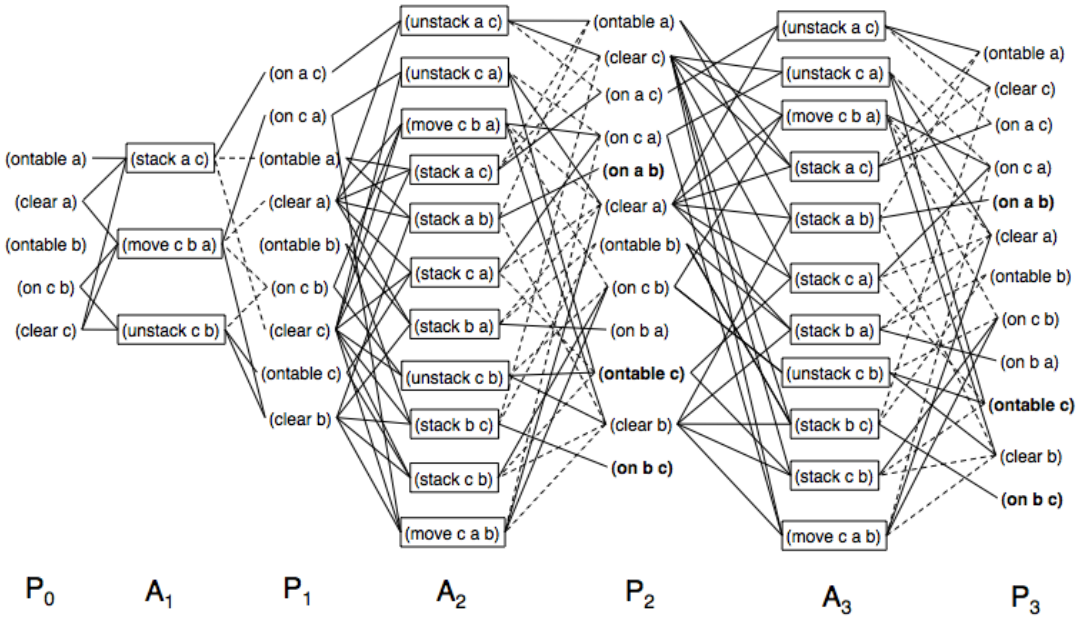


Figure 4.11: Agent *stacker*'s contribution in the Blocks World example problem.

Fig. 4.11 shows that agent *stacker* adds a new level to the *planning graph* in which all of the goal propositions are successfully achieved. Even though the goal propositions were already achieved at level $P_2$, there were *mutex* relations between them, at that level. In level $P_3$ these propositions are no longer *mutex* due to the explicit persistence introduced by `no-op` actions. In fact, if we analyse the *planning graph* we can see that the two actions that are needed in order to achieve the goal at level $A_2$, `(stack b c)` and `(stack a b)`, cannot occur at the same time at this level. The alternative is to execute these in sequence, considering however that action `(stack b c)` must occur first. So, the expansion to level $A_3$ (and corresponding proposition levels) is required to achieve all goal propositions with no *mutex* relations between them.

At this point, agent *stacker* is ready to search the *planning graph* backwards for a solution plan. The agent can either perform this task alone or request the collaboration of one of the other agents, in which case, each one will use a

different search heuristic. Either way, the agent is able to find the final solution plan, which is obviously the same as the one found in Listing 4.3.

An interesting aspect of distributing the *Graphplan* algorithm is evidenced by the difference between the final planning graphs produced by the centralised version (in figure 4.7) and the distributed version (in figure 4.11). The distributed version has produced a smaller planning graph, which is a consequence of the partial contributions approach. That is, agents add elements to the *planning graph* as needed, as opposed to the centralised approach in which the single agent that has all the necessary knowledge will try to use it all at once. In fact, for this particular example, in which only actions `stack/2` and `unstack/2` are actually used in the final plan, if agent *mover* was not called upon to participate in this problem solving process, the *planning graph* produced by the two agents (*stacker* and *unstacker*) would be even smaller and thus faster to generate. However, one must still account for the overhead introduced by the agent discovery process and the communication inherent to a distributed approach.

### 4.2.4 Goal-directed Distributed Graphplan

In most domains, some of the propositions contained in the initial state are completely irrelevant to reach the goal state of a specific problem. Consider, for example, that we add to the Blocks World example above a pair of blocks, $D$ and $E$ (which are simply placed on the table), and leave the rest of the problem unchanged. For a person, devising a plan to this new problem or the previous one is exactly the same, since he or she would be able to determine that the two extra blocks that were added are completely irrelevant to the problem.

However, as most forward-based planners, *Graphplan* suffers from the problem of *distraction*, where the planner considers all propositions in the initial state even if they will not help reach a solution plan. For example, even though this would not be used in the solution, the planner would try all combinations of stacking/unstacking/moving blocks $D$ and $E$ with the remaining blocks while generating the planning graph. These unnecessary propositions have an undesirable effect because they can be very time-consuming, thus degrading the performance of the planner. Therefore, they should be avoided. The problem lies in the fact that

forward-chaining planners do not know which propositions are relevant to the solution.

To cope with this problem, we have used a similar approach to (Kambhampati *et al.*, 1997). We introduce *means-ends* analysis in the *Graphplan* algorithm, by first producing an *operators-graph* (Smith & Peot, 1996) using a backward-chaining process starting from the goal state. Since it only considers the propositions in the goal state, the *operators-graph* will produce a graph with only relevant actions. The challenge is, however, distributing this process.

This planner uses a similar process to the one used in the generation of the *planning graph* but in a different direction, as depicted by Algorithm 6. It finds actions (including `no-op` actions) that can contribute to propositions in the goal state (line 5 of alg. 6) and the preconditions of those actions become new goal propositions (line 6 of alg. 6). As propositions become satisfied by the initial state, they are marked as *satisfied* and that information propagates throughout the entire graph. That is, actions which preconditions are satisfied are also marked as *satisfied*, which in turn will allow marking their effects as *satisfied* and so on.

This process carries on until the *operators-graph levels-off* (line 9 of alg. 6), in which case it is forwarded to another agent, (line 10) or all goal propositions are marked as *satisfied* (line 12). In this case, the forward expansion of the *planning graph* (Algorithm 3) can take place (line 13 of alg. 6), except this time it considers only the actions that are currently contained in the *operators-graph*, thus significantly reducing the size of the *planning graph* and the number of *mutex* calculations. Note that the `Expand'` algorithm used here is slightly different from the `Expand` algorithm (Algorithm 3). Instead of forwarding the *planning graph* when it *levels-off* (as in line 12 of alg. 3), `Expand'` returns `null`.

The actions generated by the *operators-graph* might not be enough to create the solution plan, in which case a new level in the graph is created (lines 14 and 15 of alg. 6), possibly generating new actions to be considered.

This alternate process carries on until either the solution is found or the *operators-graph levels-off* (line 9 of alg. 6). Since there are no mutex*es* in this process, the condition associated with a *leveled-off* graph is slightly different: the

**Algorithm 6** `BuildOG(i, OG)`: Let `i` be the current level of expansion in the *operators-graph*, `G` the set of goal propositions, `OG` an *operators-graph* with the structure $\langle P_n, A_n, \ldots, P_1, A_1, P_0 \rangle$, $PG$ a *planning graph* with the structure $\langle P_0, A_1, \mu A_1, P_1, \mu P_1 \ldots, A_n, \mu A_n, P_n, \mu P_n \rangle$ and `A` the set of actions the agent knows:

1: **if** $i = 0$ **then**
2: $OG\langle P_0 \rangle \leftarrow G$
3: `BuildOG`$(1, OG)$
4: **end if**
5: $A' \leftarrow \{a \in A, a \notin A_i, p \in P_{i-1} \mid (effects^+(a) \vdash p\}$
6: $P' \leftarrow \{p \mid \exists\ a \in A' : p \in precond(a)\}$
7: $A_i \leftarrow A_i \cup A'$
8: $P_i \leftarrow P_i \cup P'$
9: **if** $|P_{i-1}| = |P_i|$ and $|A_{i-1}| = |A_i|$ **then**
10: `AnalyseAndForward`$(OG)$
11: **else**
12: **if** $\{\forall\ g \in G,$ `satisfied`$(g)\}$ **then**
13: $PG \leftarrow$ `Expand'`$(0, \oslash, OG)$
14: **if** $PG = \oslash$ **then**
15: `BuildOG`$(i + 1, OG)$
16: **else**
17: *return* $PG$
18: **end if**
19: **else**
20: `BuildOG`$(i + 1, OG)$
21: **end if**
22: **end if**

*operators-graph* has a fixed-point level `k` that is the smallest `k` such that $|P_{k-1}| = |P_k|$ and $|A_{k-1}| = |A_k|$.

The generation of this graph is faster because it is not as complex as the forward *planning graph* generation (which includes calculating *mutexes*). But since it does not analyse the relations between actions of the same level, it still generates actions that, even though relevant, cannot occur in a solution plan. Nevertheless, this approach still presents advantages for domains in which the *distraction* problem has an important negative impact, because it considers a lot less actions than the original *Graphplan* algorithm. The drawback is, obviously, the overhead introduced by the generation of the *operators-graph*.

Let us consider the Blocks World example again in the same distributed setting

as in the previous section, but this time the algorithm used will be Algorithm 6. Agent *stacker* is the first receiving the problem to be solved. After processing it using alg. 6, the agent creates the *operators-graph* depicted by Figure 4.12.

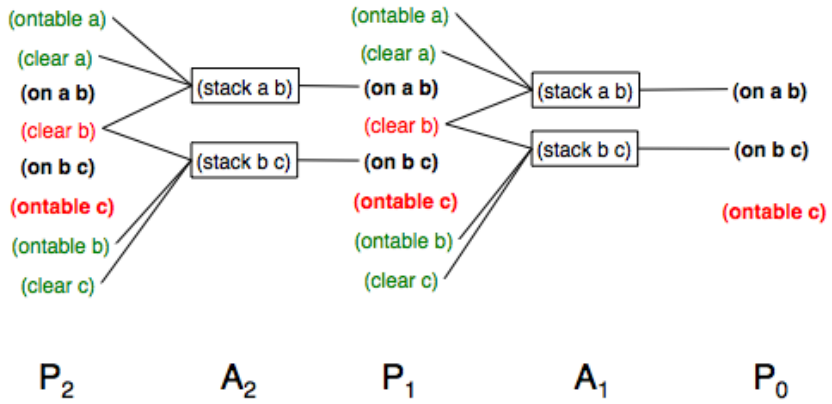


Figure 4.12: Agent *stacker*'s contribution to the *operators-graph* in the Blocks World example problem.

Starting from the goal propositions, agent *stacker* adds actions `(stack a b)` and `(stack b c)` to the graph since these can contribute to achieve, respectively, the goal propositions `(on a b)` and `(on b c)`. `No-op` actions are also added but they are not represented in the figure. The goal proposition `(ontable c)` remains unsatisfied (represented in red in the figure) as this agent does not have the necessary action to contribute to it. The preconditions of the two actions added now become new goal propositions to be achieved. Some of them can be satisfied by the propositions in the initial state (the ones represented in green in the figure) but proposition `(clear b)` remains unsatisfied.

Agent *stacker* performs a new expansion of the graph but this only causes it to *level-off*. Faced with the impossibility to further contribute to the problem, the agent must now find another agent that can perform the necessary contributions. Starting from the unsatisfied goal propositions and using the information in the *semantic overlay network*, agent *stacker* determines that agent *unstacker* can contribute to the goal proposition `(ontable c)`. After receiving the partially-filled *operators-graph* and analysing it by using alg. 6, agent *unstacker* completes

the graph as described in Figure 4.13.

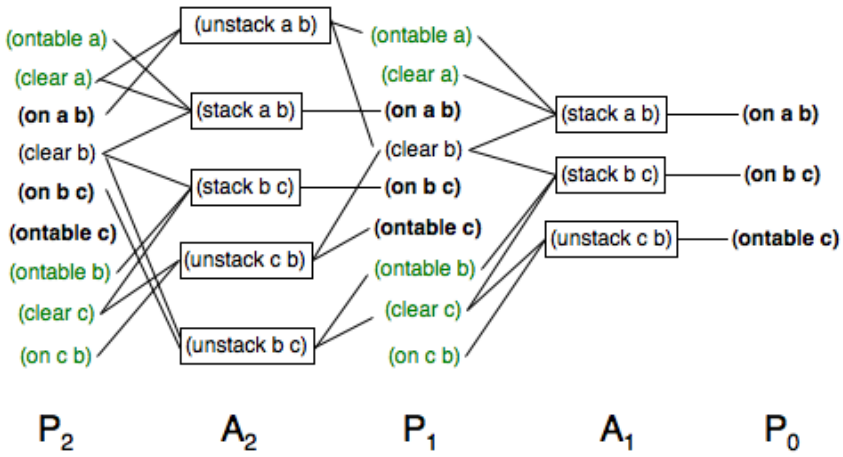


Figure 4.13: Agent *unstacker*'s contribution to the *operators-graph* in the Blocks World example problem.

In order to contribute to goal proposition `(ontable c)`, agent *unstacker* knows it can use the partially-instantiated action `(unstack c ?b)`. Note that `?b` is a variable and does not necessarily refer to block `b` in the example. This partial instantiation is obtained from the goal proposition `(ontable c)` and the action's definition, which states that action `(unstack ?a ?b)` produces the effect `(ontable ?a)`. Instantiating `?a` with `c` in the goal proposition also instantiates the action as `(unstack c ?b)`. The agent now has to calculate all possible instantiations of this action using the information already contained in the graph and with the propositions in the initial state. At this point, the only possible instantiation is the one caused by the proposition in the initial state, `(on c b)`. Since action `(unstack c ?b)` has precondition `(on c ?b)`, which unifies with `(on c b)`, the agent can fully instantiate the action as `(unstack c b)` and add it to the graph.

The agent then proceeds to analyse the second level in the *operators-graph* as some propositions are still unsatisfied. Adding action `(unstack c b)` to the second level ($A_2$) causes proposition `(clear b)` to become satisfied, thus leaving no more open conditions. This means, the forward expansion process can start

and, as such, the *planning graph* in Figure 4.14 is produced.

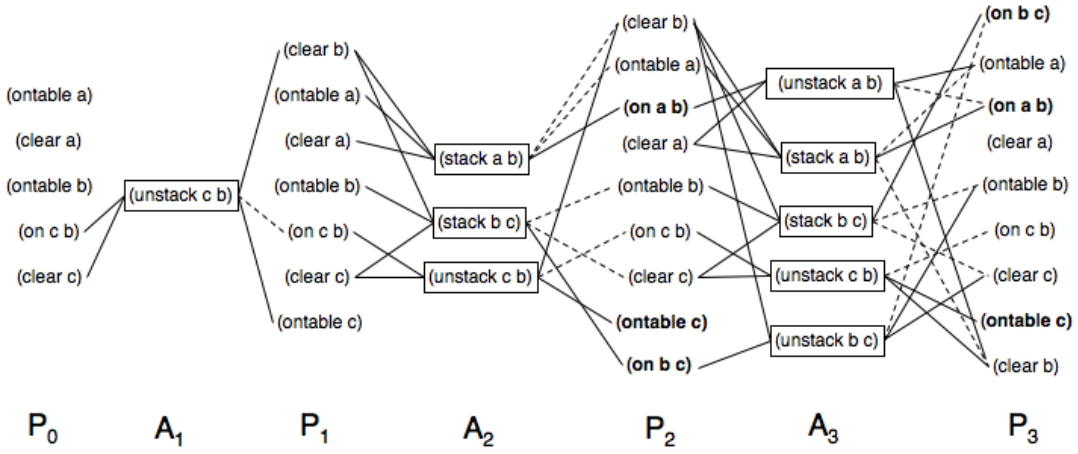


Figure 4.14: Planning graph generated by using the *operators-graph* in the Blocks World example problem.

This *planning graph* only uses the actions generated by the *operators-graph*, which in this case means the *planning graph* is a lot smaller than previously seen with other approaches. However, in order to save time in the *planning graph* generation process, the agents must first spend some time generating this *operators-graph*. Hence, the introduced overhead of generating the *operators-graph* must not exceed the time it takes to normally generate the *planning graph* in order for this to be considered as a valid alternative.

Not every situation may take advantage of this backward *operators-graph* generating process. In those domains where almost all propositions in the initial state are relevant to building the solution this process will hardly bring any advantage. Also, as we have seen above, this process includes a potentially time-consuming step of calculating all possible combinations of partially-instantiated actions. This step leads to an exponential branching in the graph generation process therefore, in domains where there are actions that have a lot of parameters, there is a high probability of this step leading to considerable degradation of the overall performance. However, in most domains, this approach may bring just enough improvement to justify its inclusion in the distributed planning process.

## 4.3 Substantiating Decisions

Throughout the research work, and especially through the conception of our technical proposal, some decisions were made regarding the approaches or tools to use in certain parts of the system. This section focuses on substantiating those decisions by summarising the existing alternatives and explaining, through empirical evidence, the final decision.

### 4.3.1 Network Evolution Techniques

In section 4.1.3, we discussed some adaptation procedures intended to help agents improve their searches over time. These procedures depend mainly on information gathered during previous searches and on information shared by agents. Using those procedures, agents can acquire useful knowledge regarding the location of other agents and resources and thus contribute to the evolution of a better-adapted network.

One of the procedures includes caching information regarding previous searches so as to have useful information in case the same searches are triggered. Basically, information about the location of a certain agent or resource is stored and can be used as a referral for future searches. Searches that are based on this acquired knowledge are called *informed* searches. However, agents can only store information that is passed through them as the search query propagates through the network. That is, if they are not on the path travelled by the search query, they do not acquire the referred information.

Even though exchanging this information may be useful for agents to improve future searches, it also immensely increases their communication load. This leads us to an important decision: once an agent has the answer to a query, should it reply through the same path that was travelled by the query, thus allowing the participating agents to learn this answer and store it for future reference? Or should it reply directly to the requester agent (which in turn receives the answer faster) avoiding an excessive communication load but also preventing the participating agents from learning the answer?

In order to decide which approach should be taken, we have performed tests with several different configurations of search algorithms in the dynamic creation

of the *semantic overlay network*. In these tests, all agents start searching for the skills that they depend on at the same time. We use the term *network completeness* to represent the percentage of the *semantic overlay network* that is created at a specific moment. The *semantic overlay network* is complete when all agents have established a dependency connection for all of their own skills.

We have performed these tests with different algorithms (Depth-First Search (DFS), Flooding, IBDFS and PbF) and all of them have shown similar results. We show only the results for the *Flooding* algorithm, for which Table 4.1 presents the different configurations that we have used in the tests.

| **Name** | **Description** |
|---|---|
| *Flooding* | Agents do not cache previous searches and reply through the path where the request came from |
| *Improved Flooding 1* | Agents do not cache previous searches and reply directly to the requester agent |
| *Improved Flooding 2* | Agents cache previous searches, use referrals and reply through the path where the request came from |
| *Improved Flooding 12* | Agents cache previous searches, use referrals and reply directly to the requester agent |

Table 4.1: Configurations of the *flooding* algorithm

These tests were performed in an environment of 1000 agents, randomly connected to 3 neighbours each and with a *time-to-live* parameter of 3 (each request can only be forwarded 3 times). As depicted in Figure 4.15, the classical configuration of the *flooding* algorithm has the worst performance of all configurations, achieving a *network completeness* of only 20%.

The *improved flooding 1* configuration, representing the version of the algorithm that allows agents to reply directly to the requester agent (instead of using the request path), presents an improvement in time performance whereas the network completeness is maintained at 20%. This allows us to conclude that the reduction in the number of messages (in consequence of the introduced variation) and consequently on the workload of each agent is a good network evolution technique to be applied to a search algorithm. However, as stated above, this does not allow all contributing agents to learn the response to the request.

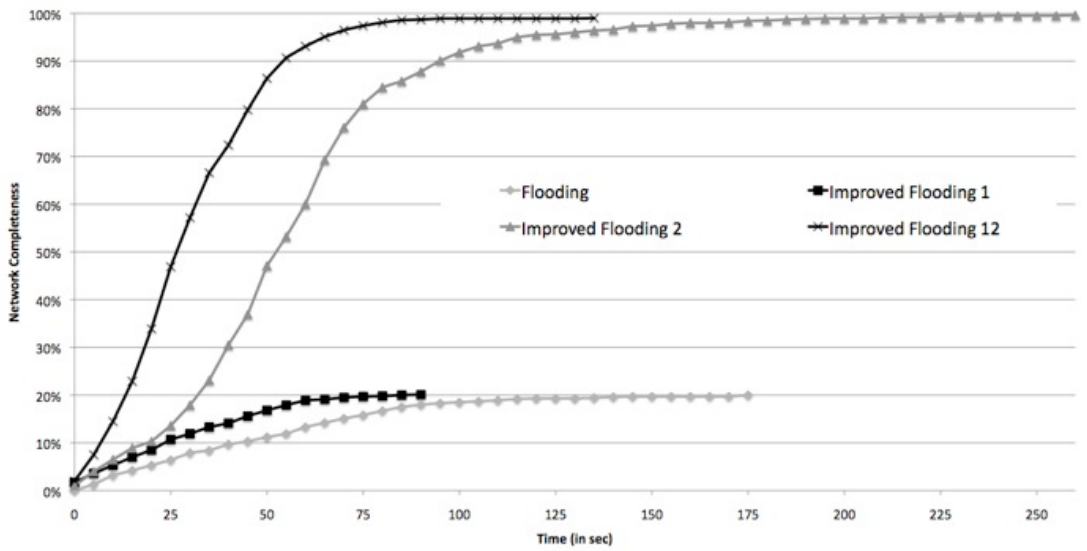


Figure 4.15: Comparison of different configurations of the Flooding algorithm

The *improved flooding 2* configuration, representing the version of the algorithm that allows agents to cache information about previous searches and use referrals, has the worst time performance but a considerably better network completeness than the previous two configurations of the algorithm. This shows that caching has also a positive effect on the search algorithms overall performance, since it allows agents to take advantage of previously collected information to trigger an evolution process that will improve future searches.

Finally, the *improved flooding 12* configuration presents an excellent performance (comparatively to the other configurations) both in time and network completeness. Even though in this configuration, agents reply directly and do not learn as much as in the *improved flooding 2* configuration, we can see that the combination of reduced communication load and cached referral information has a very positive influence on the efficiency of the algorithm. Hence, the decision here was to use the approach represented by this configuration in all the other search algorithms presented in the thesis.

### 4.3.2 Planning Algorithms

The work described in section 4.2 is based on a well-known planning algorithm, *Graphplan*. The decision to use this planner in our approach was based on the flexibility and innovative characteristics of the *Graphplan* algorithm that allowed it to be considered as one of the fastest planners ever created. In fact, the International Planning Competition, a biennial contest associated with the International Conference on Automated Planning and Scheduling, has been the stage for showing how planning graph-based planners have evolved and presented excellent results in different domains and areas. In order to demonstrate the superiority of this planner, we have devised a simple test to evaluate the performance and quality of *Graphplan* and several other planners.

In total, we used four different planners[8] that represent different approaches to classical and neo-classical planning. The two first planners represent the total-order state-space planning approach (see section 2.1.1 for details). One uses a forward-chaining approach (TOFC), in which the planner adds actions to the solution plan starting from the propositions in the initial state, and the other uses a backward-chaining approach (TOBC), in which the planner adds actions to the solution plan starting from the goal propositions instead.

The third planner represents the partial-order plan-space planning approach (see section 2.1.2 for details). While state-space planning views a plan as a strict sequence of actions that achieves a desired goal state, plan-space planning, provides a more complete view of a planning problem by introducing the notion of partially specified plans. A partial plan can be viewed as a structured collection of actions that provides their causal relationships, as well as their intrinsic ordering and variable binding constraints. The partial-order planner (POP) iteratively analyses the current partial plan and adds new actions or constraints accordingly, ultimately producing a complete and consistent solution plan that defines a path from the initial state to a state containing all goal propositions.

The fourth planner is the *Graphplan* algorithm, which represents the planning graph approach (see sections 2.1.3 and 4.2.2 for details). We implemented centralised versions of all of these planners using the same programming language

[8]All planners were implemented by us using the the JAVA™ programming language and the formal descriptions presented in (Ghallab *et al.*, 2004).

and performed the tests using the same machine, so as to avoid any discrepancies in the evaluation process. The tests consisted on a set of increasingly larger planning problems in the Blocks World. Basically, the planners have to produce solution plans for reversing sets of 3-block piles and, on each new problem, the number of 3-block piles is increased by one. Figure 4.16 shows the different problems that were used in the evaluation of the planners.

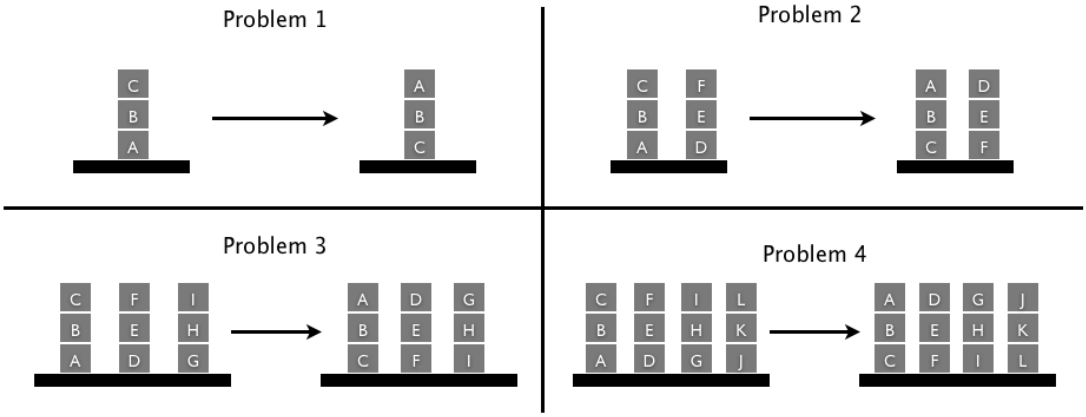


Figure 4.16: 3-block piles problems used in the tests

We tested all of the planners against these problems and evaluated them in terms of performance (time taken to present a valid solution plan, measured in seconds) and quality of the solution plan (number of steps in the solution plan relative to the optimal solution, measured in percentage[9]).

Figure 4.17 presents the results of the tests for both measurements. Regarding performance (left diagram in the figure), the superiority of the *Graphplan* approach is quite clear. The performances of all the remaining planners rapidly degrade as the size of the problems increases, whereas *Graphplan* maintains a stable performance. The backward-chaining version (TOBC) of the totally-ordered state-space planners has the worst performance followed by the forward-chaining

[9]If the planner has reached 100% quality, it means it has produced the optimal solution. Hence, a lower percentage means that the produced solution plan is worse than the optimal solution because it has more steps (with the assumption that each step takes exactly the same time to be completed).

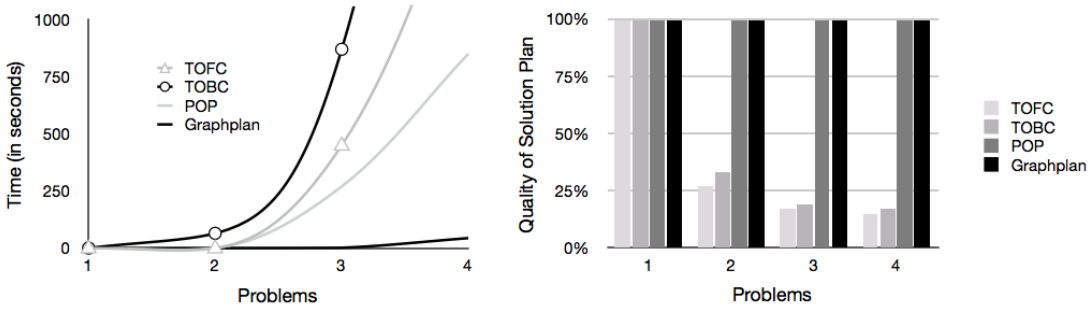


Figure 4.17: Results of the 3-block problems tests. On the left, evaluation of the performance of the planners. On the right, the quality of the solution plan relative to the optimal solution.

version (TOFC). The partial-order planner (POP) is slightly faster than the total-order planners but its performance also quickly degrades as the problem size grows.

Regarding the quality of the solution plan (right diagram of figure 4.17), all of the four planners have produced the optimal solution in the first problem, but the total-order planners are unable to produce optimal solutions for the remaining problems. This is due to the fact that these planners are focused on finding a solution and not necessarily the optimal one. This holds true for POP and *Graphplan* but, due to their properties (see details in sections 2.1.2 and 4.2.2 respectively), it is often the case that the optimal solution is the one produced. In fact, for all four problems, both the POP and the *Graphplan* algorithms have produced the optimal solution, as indicated by the right diagram in figure 4.17.

In light of these results, we have decided to use *Graphplan* as the planning algorithm in our technical approach, as described in section 4.2.

### 4.3.3 Heuristics in Planning Graph Backward Search

In section 4.2.3, we described the backward search process that is carried out to find a valid solution plan when the generation of the planning graph is complete. This search process starts from the goal propositions and finds sets of non-*mutex* actions that contribute to those goals and then the preconditions of those actions become new goals (in the previous level). This process continues until the first

level is reached successfully, in which case, pending recursions successfully return a solution plan.

Although the planning graph generation process is distributed, allowing different agents to contribute to the planning problem, this backward search process cannot be distributed. Moreover, sending the planning graph to other agents, so that these could also perform a backward search, would be pointless because they would simply be duplicating efforts.

However, as depicted by Algorithm 5, this search process has two important choice points that may affect the performance of the search process: choosing a goal proposition (line 9 of alg. 5) and choosing an action resolver (line 14 of alg. 5). This could be used as a way to distribute the backward search through different agents as well.

Still, this would not be a "*divide and conquer*" approach. Instead, all agents would be working on the same planning graph but each one would be using a different heuristic. This can be thought of as a maze with multiple entrances. The path to the other side of the maze constitutes the solution plan. The goal is for at least one agent to find the solution, which it can then share with the others. If each agent starts at a different entrance, chances are they will arrive at the other side at different times because some paths take less time to travel than others.

The difference is that travelling through a maze is a totally uninformed task, whereas the algorithm for the backward search can be focused with heuristics for selecting the next proposition `g` in the current set `G` and for choosing the action `a` in `resolvers`. A general heuristic consists of selecting first a proposition `g` that leads to the smallest set of `resolvers`, that is, the proposition `g` achieved by the smallest number of actions. For example, if `g` is achieved by just one action, then `g` does not involve a backtrack point and it is better if it is processed as early as possible in the search tree. A symmetrical heuristic for the choice of an action supporting `g` is to prefer `no-op` actions first because they have less preconditions.

Other heuristics that are more specific to the planning-graph structure and more informed take into account the level at which actions and propositions appear for the first time in the graph. The later a proposition appears in the

planning graph the most constrained it is. Hence, one would select the latest propositions first.

Considering all these possibilities, we decided to analyse the effect that different heuristics have on the performance of the planner and particularly, on the backward search process. We tested all possible combinations of heuristics for choosing a goal proposition and heuristics for choosing resolver actions. The following is a list of the heuristics for choosing a goal proposition:

- **FIFO** – priority to propositions that appear earlier in the graph;
- **LIFO** – priority to propositions that appear later in the graph;
- **Res$^-$** – priority to propositions that have fewer action resolvers;
- **Res$^+$** – priority to propositions that have more action resolvers;
- **Random** – propositions are randomly chosen;

The following is a list of the heuristics for choosing a resolver action:

- **Precond$^-$** – priority to resolvers that have fewer preconditions;
- **Precond$^+$** – priority to resolvers that have more preconditions;
- **Random** – resolvers are randomly chosen;

The tests consisted on running the planner 10 times for each possible pair of heuristics on a planning problem for the Blocks World, in particular, problem 4 described in section 4.3.2. Table 4.2 presents the average results for all possible pairs of heuristics. Rows represent heuristics for choosing a resolver action and columns represent heuristics for choosing a goal proposition.

The results seem to support the hypothesis presented above, that is, choosing resolver actions that have less preconditions (which has a lower impact if backtracking occurs) and choosing propositions that appear later in the graph (which are more constrained) has a very positive effect. That combination (Precond$^-$ with LIFO) had the best time performance of all possible combinations (44 ms) and, in general, these individual heuristics combined with other heuristics (see

| | **FIFO** | **LIFO** | **Res$^-$** | **Res$^+$** | **Random** |
|---|---|---|---|---|---|
| **Precond$^-$** | 45 | **44** | 240 | 5645 | 69 |
| **Precond$^+$** | 90 | 49 | 230 | 8092 | 315 |
| **Random** | 67 | 52 | 242 | 6790 | 298 |

Table 4.2: Comparison of heuristics in *Graphplan* backward search process. Values are in milliseconds and represent *only* the time spent in the backward search process

row Precond$^-$ and column LIFO) have also presented satisfying results compared to other combinations.

Processing the number of resolvers that a goal proposition has in order to choose the proposition with fewer resolvers (column Res$^-$), while it could apparently have a positive effect, the time spent determining the proposition with fewer resolvers is too much to actually bring any gain compared to the LIFO approach. Also, it is quite clear that using the opposite approach (column Res$^+$) severely affects the performance of the search process, due to the "heavy" backtracking that is required to deal with giving priority to goal propositions that have more resolvers.

Random heuristics do not present any generic pattern, in some cases presenting good results and in others the worst results, which is consistent with the random choice of goal propositions and resolver actions and further proves the results are sound.

Based on these results, we have decided to use the Precond$^-$ and Precond$^+$ heuristics for choosing resolver actions and the FIFO and LIFO heuristics for choosing goal propositions. This way, each agent participating in a backward search process can use a different combination of heuristics. We also have to consider that these results may be different for more complex problems (like the ones presented in chapter 3), which further motivates the use of different heuristics and the participation of different agents in the backward search for a solution plan.

# Chapter 5

# Evaluation

Our main goal, as stated in the introductory chapter, was to develop a robust and scalable approach that enabled agents to efficiently participate in distributed problem solving in unstructured agent societies. Although the technical work described in this thesis was based on solid theoretical research, there is still need to empirically prove the resulting system is satisfactory according to the goals that were set. In this chapter we describe the evaluation process that was carried out in order to determine if our approach is indeed robust, efficient and scalable.

First of all, let us clearly define each one of these terms so as to avoid any confusion in the interpretation of the results. The efficiency of a system can be measured in many different ways, which prompts for an exact definition of a satisfactory evaluation process to measure a system in terms of its efficiency. In general terms, an efficient system is expected to be fast (return results in useful time) but it is also expected that the usage of available resources to accomplish its goals is the best possible. Thus we consider these two dimensions when comparing our system against the alternatives, in terms of efficiency.

Robustness refers to the capacity of a compound system to withstand any failures or poor functioning of its composing elements. We aim to evaluate, from this point of view, the impact on the operation of the system of removing or disabling some of its elements.

Finally, scalability refers to the capacity of a system to maintain the same level of performance as the problem/domain/environment grows. To this end, we intend to evaluate the behaviour of the system (and corresponding alternatives)

by increasing the size of the problem that it needs to solve. It is then expected that the performance of the system does not decrease more than proportionally with the problem size. Ideally, although the overall performance degrades (hopefully not more than proportionally) with problem size, the system should still improve its performance over time.

Since our system is composed of two different independent parts (planning and agent discovery), we decided to separate the evaluation process in two. First, section 5.1 describes the tests and conclusions of the evaluation performed to the agent discovery process and the generation of the semantic overlay network. In this case, no particular planning approach is considered. The performed tests aim to evaluate the efficiency, scalability and robustness of our approach. The efficiency, as stated above, is evaluated by comparing the performance of the algorithms in terms of speed (see sub-sections 5.1.1 and 5.1.2) and resource usage (see sub-sections 5.1.3 and 5.1.6). The robustness is measured by comparing the performance of the algorithms in terms of their capacity to withstand random failures in the network (see sub-section 5.1.4). The scalability is measured by comparing the performance of the algorithms when the size of the network is increased (tests shown in sub-section 5.1.5).

Section 5.2 describes the tests that were performed to the overall distributed problem solving approach, by evaluating the performance of different approaches while solving problems based on the scenarios described in Chapter 3. In particular, we aimed to evaluate the efficiency and the scalability of our distributed planning approach by measuring the performance of the algorithms as the problems increased in size (see sub-section 5.2.1), as the distribution of skills varied (see sub-section 5.2.2) and by testing different strategies to choose appropriate resolver agents (see sub-section 5.2.3). Although the focus of this second section is on the distributed planning process, the agent discovery process and the semantic overlay network generation process are also included (see sub-section 5.2.2).

Finally, section 5.3 summarises the obtained results and outlines the major conclusions.

## 5.1 Algorithms for Distributed Agent Discovery and Semantic Overlay Network Generation

In unstructured networks, agents cannot rely on central repositories to find other agents or resources required to solve a specific problem. They have to use algorithms that rely on propagating the search queries through the distributed network. In fact, as explained in section 4.1.3, agents can use this information to ultimately build a *semantic overlay network* that establishes important semantic links between agents.

The way agents build the *semantic overlay network* and the time it takes to be generated depends on the search algorithm they used. We have proposed two different search algorithms: the Priority-based Flooding (see details in sub-section 4.1.1) and the Iterative Branching Depth-First Search (see details in sub-section 4.1.2). In this section, we present the tests that were performed in order to assess the quality of these algorithms in terms of robustness, efficiency and scalability.

In order to perform the evaluation, we need to compare the performance of these algorithms against the alternatives. We decided to use the best configuration of the *Flooding* algorithm (as explained in sub-section 4.3.1), the *Improved Flooding 12* configuration, as a reference for comparison[1]. From this point on, we will only refer to this algorithm as *Flooding* for simplicity.

The environment in which we tested these algorithms consisted of a real network of 1000 agents that were randomly connected at boot time. In general, agent networks or agent societies are made of agents with certain capabilities that can only be enacted on certain conditions. The generation of the semantic overlay network consists of each agent discovering the agents on which they depend and create virtual links to them. To evaluate this process, we simulate it in the following way. Each agent has a specific resource identified by a number. In order to build the *semantic overlay network*, each agent has to find the agent holding the resource with the number immediately before its own number (referred to as the agent it "depends" on). For example, the agent holding resource 6 has to find the agent holding resource 5. To complete the circle, the agent holding

[1]This specific configuration is used because it has shown to be the best in all of the algorithms that we have tested (excluding our own).

resource 1 has to find the agent with resource 1000. While doing this, each agent is also able to find the agents to which it can contribute (*i.e.*, an agent that has the resource numbered next to its own resource) and collect information regarding other agents in the network. As explained in section 4.3.1, this information gathering process facilitates the search process inherent to the generation of the semantic overlay network.

The term *network completeness* refers to the percentage of agents that were already able to find those resources that theirs' depend on.

To compare the performance of the algorithms, we decided to use the worst possible situation: all 1000 agents try to search the agents on which they depend at the same time. We then measured the time of the process of building the *semantic overlay network*. All the results shown here represent the average data of 10 runs, *i.e.*, each test was ran 10 times and then we collected the necessary average data. The results depicted in Figure 5.1 show the comparison between the three algorithms using the following configuration:

- Number of agents: 1000;
- Number of neighbours (NN): Each agent is randomly connected to 3 other (different) agents;
- Time To Live (TTL): Each search query can only be forwarded 3 times;

As we can see in the figure, the Iterative Branching Depth-First Search (IBDFS) algorithm reaches the same level of network completeness almost always faster than the *Flooding* and the Priority-based Flooding (PbF) algorithms. The *Flooding* algorithm and the PbF algorithm have almost the same performance, except towards the end where PbF only reaches 100% of network completeness almost 40 seconds later. In fact, the performance of the PbF algorithm is consistently slightly worse than that of the *Flooding* algorithm throughout almost all of the tests that we have performed. This is caused by the fact that the advantage of choosing local search queries first (see sub-section 4.1.1 for details) is not enough to cover the overhead introduced by the processing required to sort search queries. For this reason, we will no longer show the results for the PbF algorithm and will only show the comparison between the IBDFS and the *Flooding* algorithms.

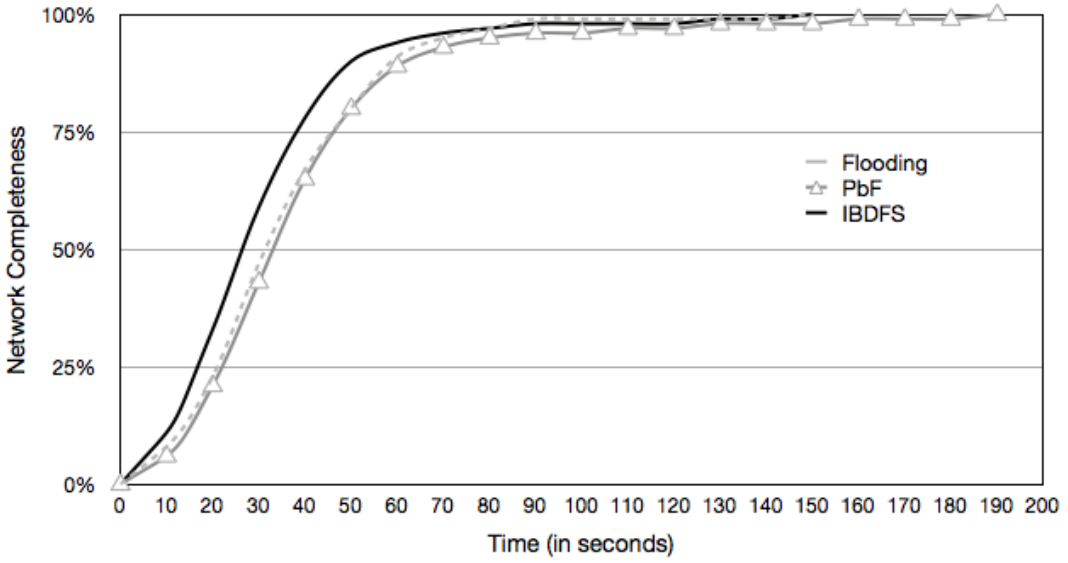


Figure 5.1: Comparison between the 3 algorithms in a semantic overlay network generation process

### 5.1.1 Variations in the "Time To Live" Parameter

To fully understand the differences between the algorithms, we changed several parameters of the test configuration and analysed the effects of those variations on the performance of both algorithms. One of the parameters that we changed was the *Time To Live* (TTL), that is, the number of times a search query can be forwarded.

Figure 5.2 shows the results of the variations of the *TTL* in the test. The left diagram refers to a TTL of 4 and the right diagram refers to a TTL of 5[2]. The other testing properties remain the same for both diagrams: 1000 agents connected to 3 neighbours each. As depicted in fig 5.2, the larger the TTL, the larger is the difference between the performances of both algorithms. Figure 5.3 presents a different view of the same test with all the configurations together in one graphic.

This figure shows that the variation in performance between the two different

[2]Although the difference between the TTL values may seem small, the effect of increasing the TTL (or the Number of Neighbours) by one unit is very significant for the performance of the algorithms, as it can be perceived in sub-section 5.1.6.

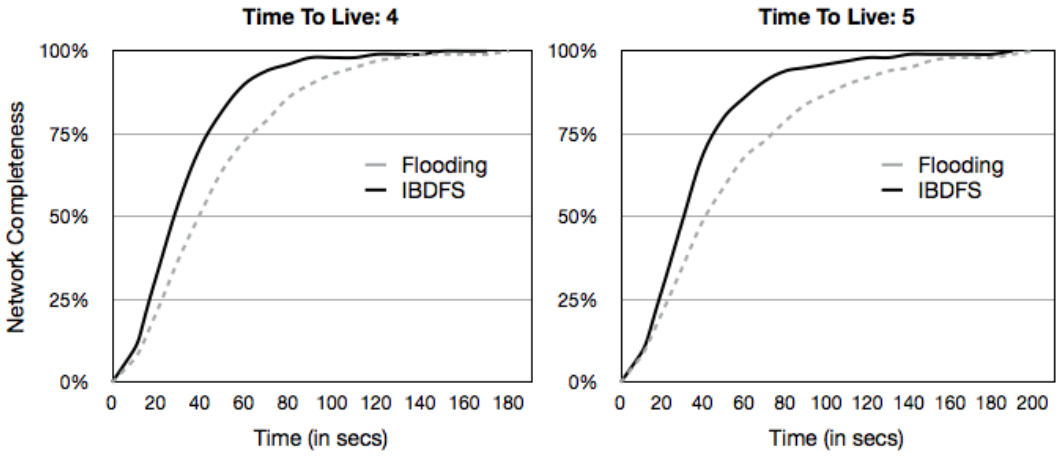


Figure 5.2: Comparison between the algorithms with a variation of the Time To Live parameter

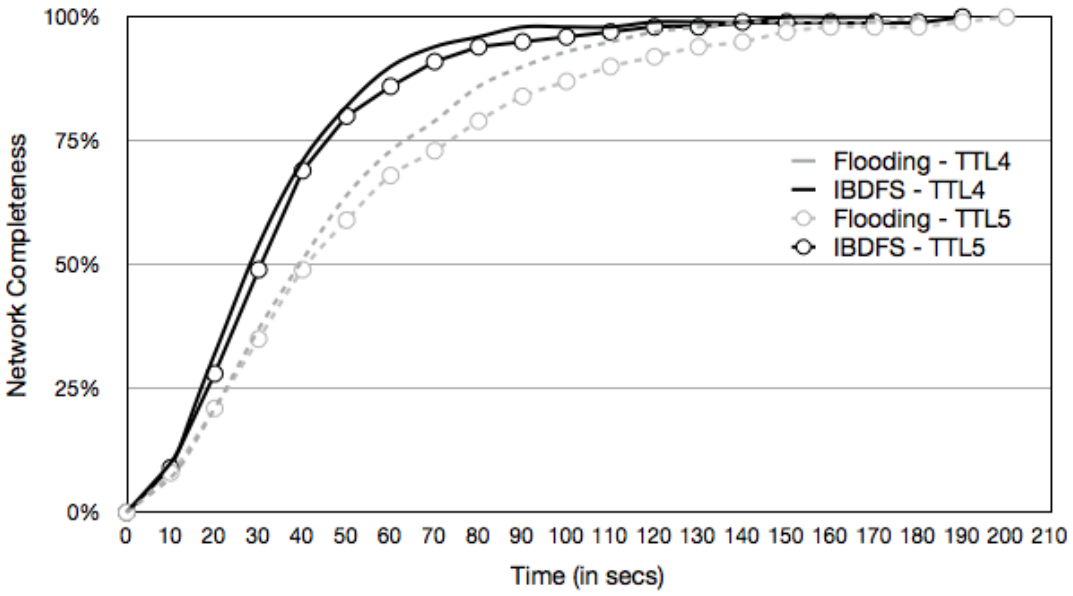


Figure 5.3: Comparison between the algorithms with a variation of the Time To Live parameter

configurations of the IBDFS algorithm is less than that between the two configurations of the *Flooding* algorithm. This suggests that the difference between the two algorithms tends to increase as the TTL parameter increases. This is due to

the overloading factor of the flooding algorithm, in which increasing TTL creates a much larger number of messages being exchanged in the network, thus delaying most search queries.

### 5.1.2 Variations in the Number of Neighbours

Another parameter that influences the performance of search algorithms is the number of neighbours that each agent is connected to when the test begins. Figure 5.4 shows the results of different tests using 4 and 5 neighbours (1000 agents and a TTL of 3). Similarly to what happened when the TTL changed, the figure shows that the difference, in performance, between the configurations of both algorithms increases as the number of neighbours increases.

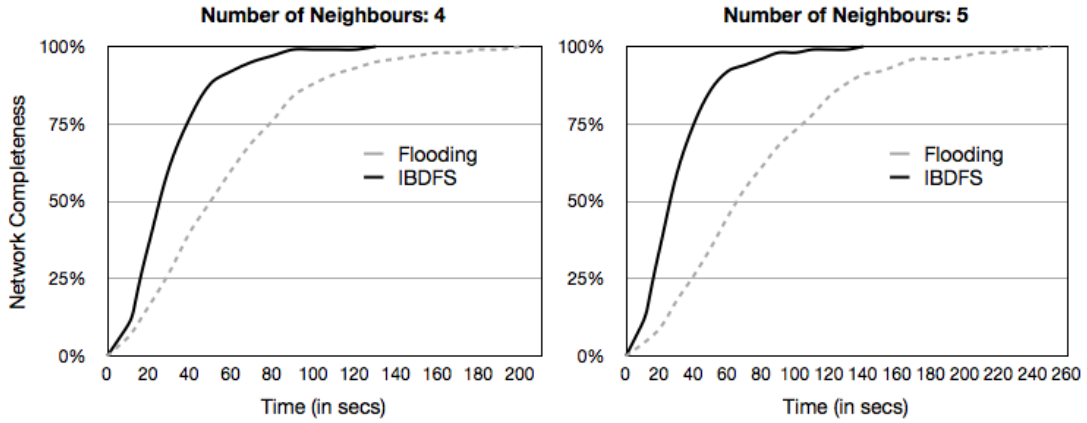


Figure 5.4: Comparison between the algorithms with a variation of the number of neighbours

Figure 5.5 presents the same test but this time with all of the configurations together in one diagram. This figure depicts a very interesting phenomenon. Increasing the number of neighbours does not seem to influence the performance of the IBDFS algorithm, whereas the *Flooding* algorithm is severely affected. This allows us to conclude that the IBDFS algorithm is well suited for high-load and high-connectivity networks. This is due to the fact that, in the IBDFS algorithm, each agent only uses the absolute necessary number of neighbours to find the

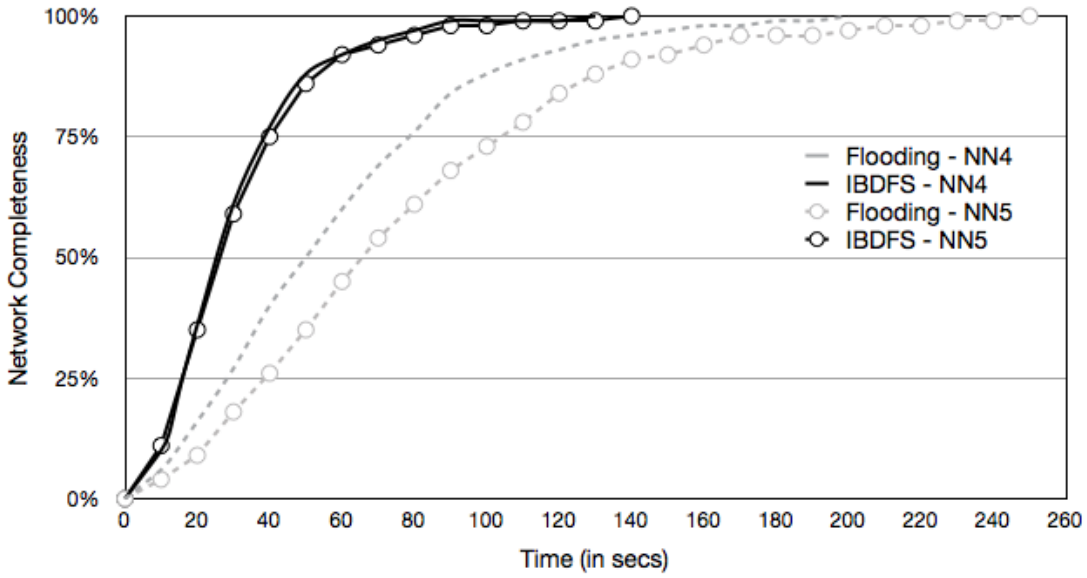


Figure 5.5: Comparison between the algorithms with a variation of the number of neighbours

answer, halting the discovery process once the answer is found. The *Flooding* algorithm uses all possible neighbours and, although using more neighbours would apparently increase the probability of finding the answer faster, it actually causes the whole network to perform poorly.

### 5.1.3 Variations in Resource Distribution

Up to now, the agents in these tests managed resources that were unique in the network, that is, each agent manages a single resource that cannot be found anywhere else in the network. To analyse how resource distribution influences the performance of both algorithms, we decided to perform the tests using different distributions of resources.

We use the expression *resource distribution factor* (RDF) as a measure of the amount of different resources existing in the network (relative to the total number of agents) and consequently their availability. For example, if the RDF is 100% (which was the case for all previous tests shown above), then the amount of resources in the network is equal to the number of agents, thus making the

resources owned by each agent unique. If the RDF is 70%, then the amount of different resources in a network of 1000 agents is 700, thus allowing multiple resources of the same type to exist in the network.

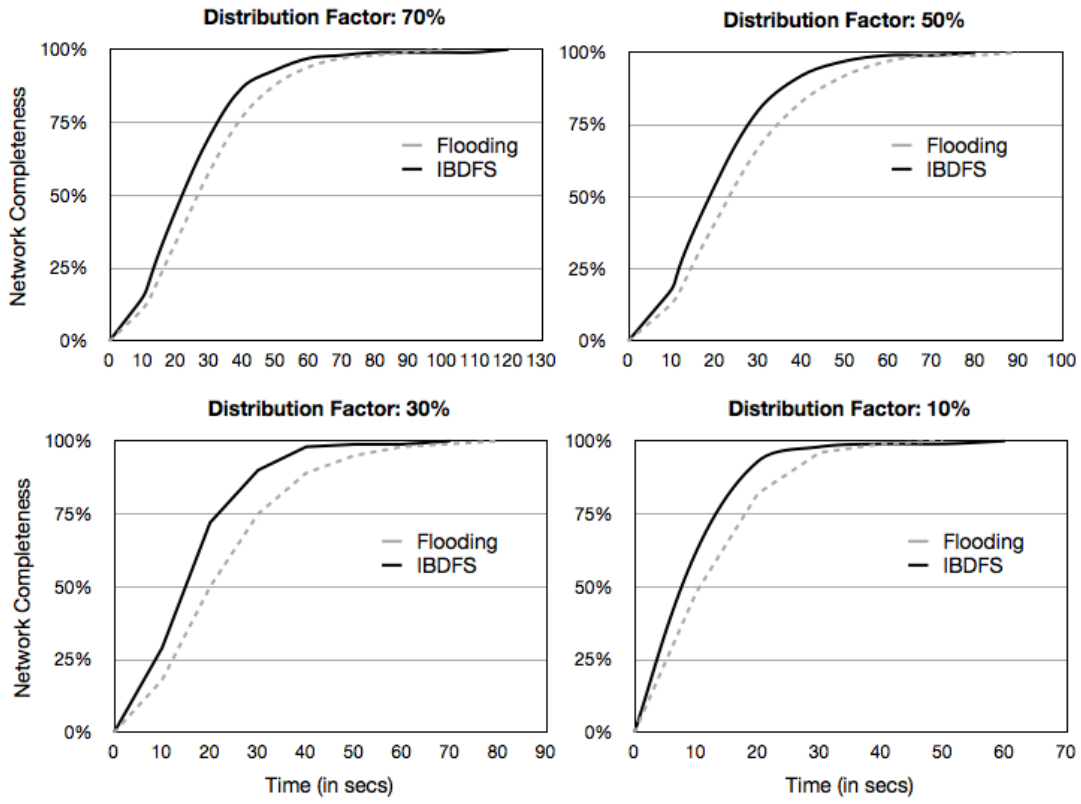


Figure 5.6: Comparison between the algorithms as the resource distribution factor changes

Figure 5.6 shows the test results for different resource distribution factors (1000 agents connected to 3 neighbours each, with a TTL of 3). As depicted in the figure, the difference, in performance, between the IBDFS and the *Flooding* algorithms slightly increases as the resource distribution factor (RDF) decreases.

However, when the distribution factor is 10%, the difference between the two algorithms seems to decrease again. This is explained by the fact that, as the distribution factor decreases, the amount of duplicate resources increases, making them very easy to find in the network. Hence, as the availability of a resource

increases in a network, the influence of a search algorithm in the time it takes to find that resource tends to decrease. Basically, there will be a point in which it does not matter which search algorithm one uses because, due to the fact that a certain resource is very common in the network and thus easily found, the search performance will always be fast.

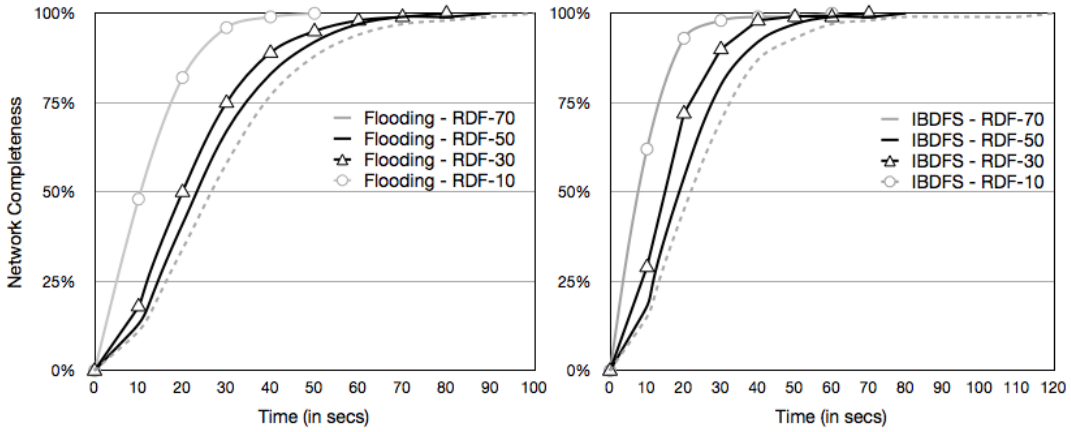


Figure 5.7: Comparison between the algorithms as the resource distribution factor changes

This can be seen in Figure 5.7. The diagram on the left shows the performance difference between the different configurations of the *Flooding* algorithm. The diagram on the right shows the performance difference between the different configurations of the IBDFS algorithm. We can see that the *Flooding* algorithm suddenly improves with a RDF of 10%. Since, in the *Flooding* algorithm, each agent uses all of its neighbours to propagate the search query, the tendency is for it to improve as the resource distribution factor decreases, as explained above.

### 5.1.4 Testing Robustness

Up to this point we have only focused on the performance of the algorithms regarding the time it takes to generate the *semantic overlay network*, but there is an important aspect of real-life networks that must be evaluated as well. It is important to assess how a certain algorithm will behave under unfavourable

conditions, for example, test how the performance of a search algorithm will be affected if some of the elements in the network cease to work.

To evaluate the robustness, we ran the algorithms several times and, on each time, each agent in the network had a certain probability of going offline. To trigger this process, each agent executes a random function that decides whether or not the agent goes offline. When offline, all received messages are ignored and discarded. A similar rule governs the process of coming back online.

We decided to use the number of received messages as a trigger to execute the random function. That is, every time the agent's message count reaches a certain value it executes the function, which decides if the agent goes offline. The message count threshold is also randomly computed. The agent randomly chooses a number between 5 and 10; the next time the agent's message count reaches a multiple of that number, it triggers the random function.

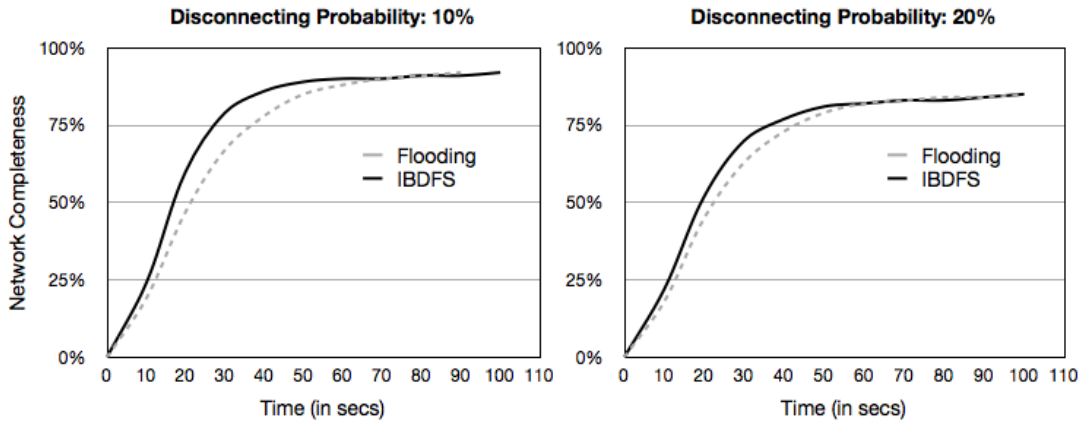


Figure 5.8: Comparison between the algorithms as the disconnecting probability changes

Figure 5.8 shows the tests done with different probabilities of going offline. The remaining parameters had the following values: 1000 agents, NN: 3, TTL: 3, RDF: 70%. Since some agents go offline, none of the algorithms is able to reach 100% of network completeness. With 10% and 20% of disconnecting probability, both algorithms are able to reach, respectively, 92% and 85% of network completeness. This is possible because the resource distribution factor is 70%,

allowing for duplicate resources in the network.

It is also visible in the figure that the IBDFS is still better than the *Flooding* algorithm. However, the difference in the performance of the algorithms seems to be decreasing as the probability of going offline increases. In fact, for probabilities greater than 28%, the *Flooding* algorithm starts to present the same performance as IBDFS. This is caused by the fact that, as more agents disconnect, each agent (when using the IBDFS algorithm) has to contact more neighbours to successfully find the necessary resource, which causes the IBDFS algorithm to introduce almost the same load on the network as the *Flooding* algorithm.

### 5.1.5 Large-Scale Networks

The evaluation done up to this point has been based on tests performed on a real network of 1000 agents. In order to assess the scalability of our approach, we needed to test the algorithms in larger networks. However, we were unable to deploy larger networks with the resources available, due to CPU and memory constraints. Hence, we used stochastic simulation based on the sequencing of a pending events chain to obtain the necessary data to evaluate the scalability of the algorithms.

Figure 5.9 presents the results of tests performed in networks of up to 5000 agents with varying connectivity (up to 5 neighbours), TTL of 5 and resource distribution (10-50%). We can see that the IBDFS algorithm outperforms the *Flooding* algorithm in every test. We can also see that the same behaviour as shown in sub-section 5.1.3 is also present here. As the resource distribution factor lowers, the algorithm's performances become closer.

Although these results were obtained in a simulation rather than on a real agent network, they remain consistent and show that the IBDFS algorithm is always faster than the *Flooding* algorithm, thus allowing us to conclude that the IBDFS algorithm is the most suitable approach to search for agents and generate a *semantic overlay network*.

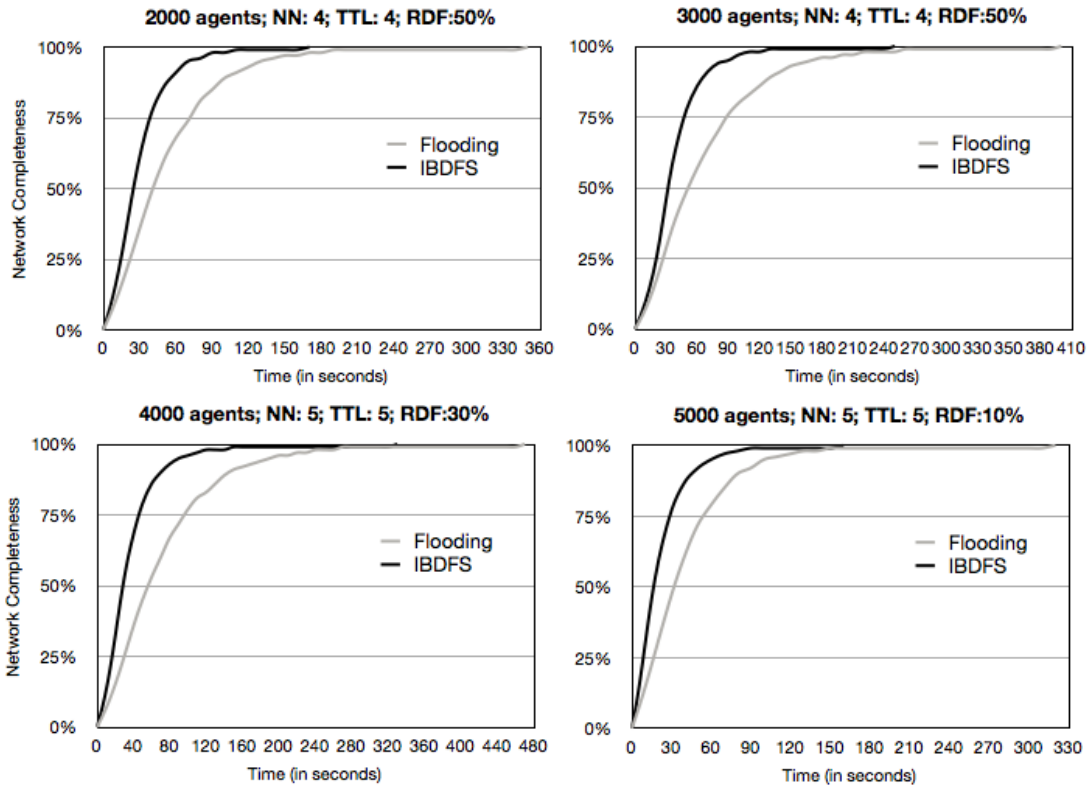


Figure 5.9: Comparison between the algorithms in larger networks

### 5.1.6 Network Load and Efficiency

In the beginning of the chapter we have set to answer the following question: does our system make better use of the available resources to achieve the same (or better) results faster than the alternatives? In fact, the IBDFS has consistently outperformed the improved version of the *Flooding* algorithm in terms of time to complete the generation of the *semantic overlay network*, as it was described in the previous sub-sections. But the question remains: has it done so efficiently, that is, has it used the available resources better than the alternative?

In this case, the available resources are the agents themselves. The agents are the ones that do all the work by processing and propagating search queries that lead to the generation of the *semantic overlay network*. So, to determine if the algorithms used the available resources efficiently we have to determine the

work load that was putted on the agents, *i.e.*, the number of messages that were exchanged.

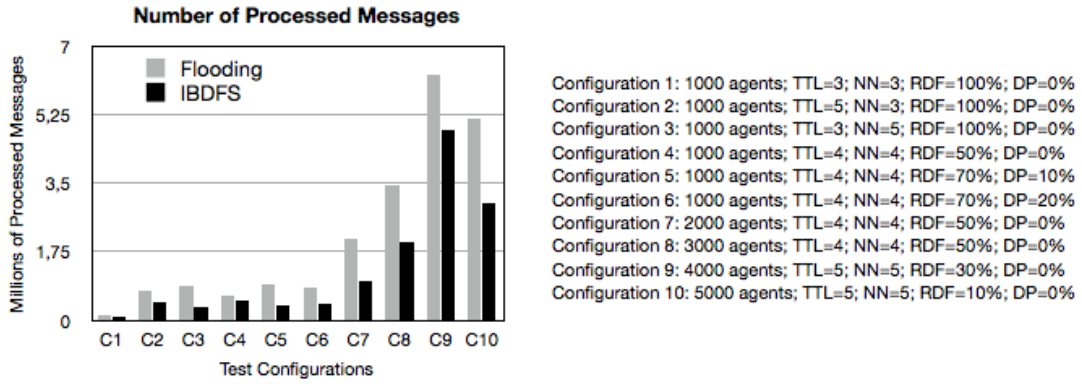


Figure 5.10: Comparison of the algorithms in terms of the number of processed messages, in different configurations

Figure 5.10 depicts a comparison of the number of messages processed by all agents in both algorithms in some of the different test configurations that were already showed in previous sub-sections. As we can see, the IBDFS algorithm has led to less messages having to be processed in all different test configurations.

The main reason for this is that the algorithm only propagates the search queries as needed, thus avoiding an excessive load on the network and allowing idle agents to deal with other search queries. Once again, in the *Flooding* algorithm, the excessive propagation and duplication of messages through the network overloads the agents with the unfruitful task of processing useless messages, which limits their capability to perform efficiently.

There is one consequence of processing less messages: agents using the IBDFS algorithm potentially learn less about other agents' skills than the agents using the *Flooding* algorithm. However, as shown by the tests in previous sub-sections, this does not seem to affect the behaviour of IBDFS, as the high connectivity of the networks (number of neighbours and TTL) is enough to compensate for the knowledge that is not being learned.

The number of processed messages can, in fact, be regarded as both a cause and a consequence of the behaviour of the algorithms in all of these tests. The

*Flooding* algorithm is based on the idea that agents should propagate the queries to all neighbours so as to obtain an answer faster, whereas the IBDFS algorithm only propagates search queries as needed (when the contacted agent replies that it does not know the answer). This has a consequence of generating more messages per agent in *Flooding* and less on IBDFS. And, while the duplication of the messages in the *Flooding* algorithm can potentially lead to finding the answer faster, it has the unwanted effect of putting too much of a workload on the agents. This then causes agents to perform poorly in comparison to a less demanding approach such as IBDFS.

In summary, the evaluation performed in this section has proven that the IBDFS algorithm is not only the fastest algorithm to generate the *semantic overlay network* (even in situations where failures can occur) but it is also the one that makes better use of the available resources by not causing the agents to process a massive number of messages. We believe that these are enough reasons to affirm that this algorithm is robust, scalable and efficient.

## 5.2 Approaches for Distributed Planning

The previous section focused on providing a detailed evaluation of the algorithms for searching and generating a *semantic overlay network*. Although important, the evaluated part is responsible only for the agent discovery and the generation of the semantic overlay network. Each agent also needs to determine how its skills can be used to build a solution for a specific problem, and to identify the sub-problems that need to be delegated to other agents. This is done by a planning algorithm that is able to find, at least, a partial plan that contributes to solve the referred problem.

As explained in chapter 4, we have developed two algorithms that are able to deal with the partial knowledge that each agent holds and combine that with the *semantic overlay network* to find other agents that can contribute to yet-unsolved parts of each received problem. One of those algorithms is our distributed version of the *Graphplan* algorithm (see sub-sections 4.2.2 and 4.2.3 for details). This algorithm behaves similarly to the original *Graphplan* algorithm but it is adapted to deal with each agent's local view of the world, thus enabling agents to

contribute to partially generate solutions for specific problems, in a distributed fashion.

The other algorithm is a goal-directed version of the distributed *Graphplan* algorithm (see sub-section 4.2.4 for details). To cope with the problem of "distraction", this algorithm first generates an *operators-graph* using a backward-chaining process starting from the problem goal propositions. Then, using only the actions generated by this *operators-graph* it proceeds in the same way as in the *Graphplan* algorithm in order to find a solution plan.

This section presents the tests performed to evaluate these two algorithms and determine which is the best approach to be used in distributed problem solving. We start by evaluating, in sub-section 5.2.1 the overall performance of the algorithms as the problems grow in size, in the two testing scenarios described in Chapter 3. These tests clearly show that the goal-directed version of the distributed *Graphplan* algorithm scales better than the other version of the algorithm, by simply employing a much more efficient planning graph generation process. Sub-section 5.2.2 continues the analysis of the scalability and efficiency of the goal-directed version of the algorithm by evaluating and concluding that its performance is not affected by variations in the number of agents and skills in the environment. Finally, sub-section 5.2.3 explores a different strategy of choosing resolver agents and evaluates how this affects the performance of the system.

### 5.2.1 Overall Performance

First of all, we wanted to test and analyse the overall performance of the planning algorithms in both scenarios, as the problems became larger. The scenarios are deliberately different to allow testing different aspects of the system. On one hand, we have the *Rescue Agents* scenario, which in spite of the low number of different types of entities, is a very complex planning scenario. In almost any situation, all entities of the environment are required to intervene to provide the best assistance possible to the injured people, thus making conflicts management the top most priority of the planning activity. Regarding the discovery process, this scenario is not such a challenge for the semantic overlay network lookup mechanism, since it is not difficult to find the appropriate capabilities in an

environment where the amount of different capabilities is so small, especially if there are many agents that have the same type of capability.

On the other hand, we have the *Custom Balls Factory* scenario, which in spite of having many different capabilities, is a fairly simple planning scenario. For each manufactured ball, only a very small set of skills is needed from the vast selection of existing capabilities, thus characterising this scenario as a discovery challenge. The planning process on this scenario only becomes relevant when the requested customisation of the ball involves a set of interdependent features requiring a specific execution sequence. For example, a ball must first be fully painted with one colour and only then can stripes be painted with another colour. If these actions were executed in reverse order then the effects of the action for painting the whole ball would cancel the effects of the action for painting stripes.

We have performed a set of tests using increasingly complex variants of these scenarios on both algorithms. In the *Rescue Agents* scenario we used 3 different types of entities (paramedic, ambulance driver and fireman) and 10 agents for each of those entities. We tested a similar problem to the one presented in Figure 3.1 of Chapter 3. We then increased the number of injured people and the number of fires (there was one fire for the tests with 1-4 injured people and a new fire was introduced on the variants with 5 or more injured people) in the environment to test the performance evolution of the algorithms.

In the *Custom Balls Factory* scenario we used 20 combinations of different types of features of the balls manufacturing process (colour, size and other distinct marks combined with painting, assembling and inflating) and 2 agents for each of those combinations. We then increased the complexity of the manufactured ball by changing the number of features of the ball and the dependencies between them.

Figure 5.11 presents the test results for both scenarios (left diagram for the *Rescue Agents* scenario and right diagram for the *Custom Balls Factory* scenario). The measured time represents the overall planning time, including the distributed graphs generation (operators-graph – where applicable – and planning-graph) and backward search. The *semantic overlay network*'s generation time is not included since it is not of significance in the overall planning time (between 100-200 ms) and because it is an activity that agents perform as they connect to the network,

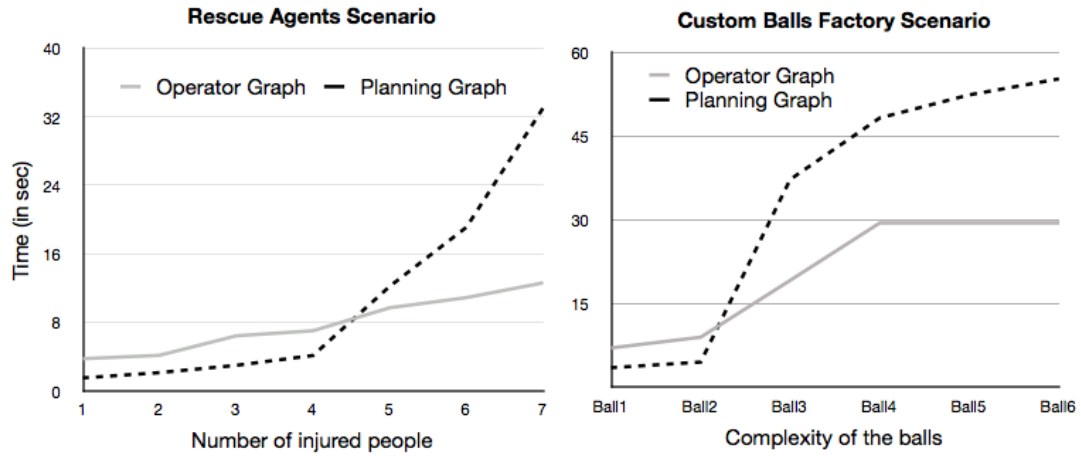


Figure 5.11: Comparison between Distributed Graphplan and Operators-graph-based Distributed Graphplan in both testing scenarios.

which means the overlay network is already built when they receive the problem solving request.

Both scenarios' test results show a similar behaviour: although the *operators-graph* based algorithm has poorer performance in smaller problems (when there are less injured people or the balls are less complex), it is clear that it scales far better than the distributed *Graphplan* algorithm. This is strongly linked to the fact that, for more complex or large problems, *means-ends* analysis is effective in reducing the planner search space, in spite of the introduced overhead. This is particularly evident in the *Rescue Agents* scenario.

The non-linear behaviour of the *operators-graph* based algorithm in the *Custom Balls Factory* scenario, apparent in the right diagram of fig. 5.11, is due to the fact that this scenario is more sensible to changes in planning complexity. As explained above, this scenario is more of a discovery challenge and, since the *semantic overlay network* is such an efficient agent discovery mechanism, as long as the number of conflicts between capabilities does not increase (e.g. due to ordering or dependency constraints), the performance remains the same. This is clear in the figure for balls 4, 5 and 6, which in spite of having a different number of features, the constraints between them are the same and thus, do not affect the performance of the system.

In order to fully understand what is really causing these behaviours in the algorithms, let us analyse a breakdown of the activities of each algorithm in the *Rescue Agents* scenario.

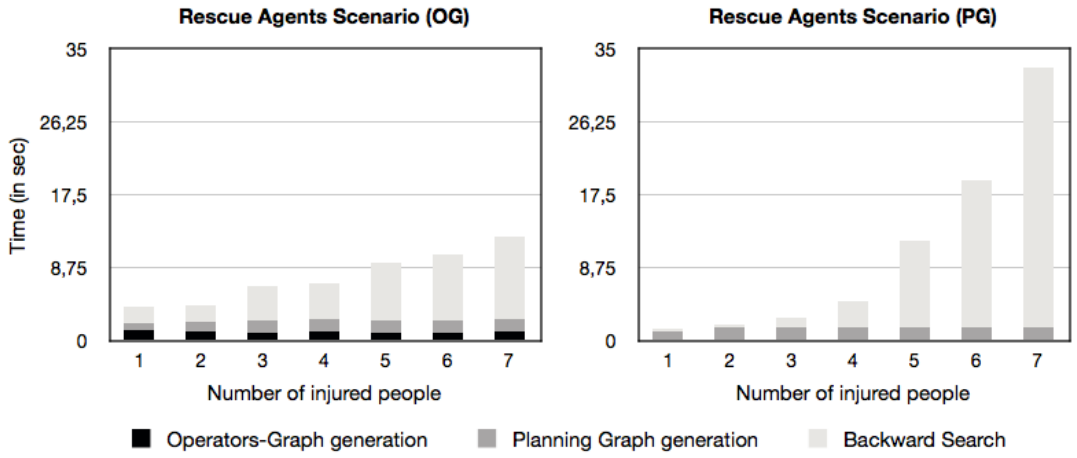


Figure 5.12: Breakdown and comparison of activities between Distributed Graphplan and Operators-graph-based Distributed Graphplan in the *Rescue Agents* scenario.

Figure 5.12 presents the breakdown of activity data for the same test as shown in the left diagram of fig. 5.11 but divided into two diagrams (the one on the left presents the *operators-graph* based algorithm and the one on the right presents the planning graph version that does not use an *operators-graph*).

As we can see, the most time-consuming activity is the Backward Search process. This is the task that involves searching the planning graph backwards in order to find a valid solution plan. The generation of the *operators-graph*, although causing poorer performance in simpler problems, is very efficient in improving the Backward Search phase in larger and more complex problems by significantly reducing the number of actions that are considered in the planning graph generation process[3].

[3]Although not shown here, the same conclusions apply to the *Custom Balls Factory* scenario

### 5.2.2 Distribution of Skills

In the previous section, we tested the behaviour of the planning algorithms as the problems became larger in order to assess their scalability and efficiency. This has shown that the *operators-graph* based version of the distributed *Graphplan* algorithm scales far better. However, the scalability and efficiency analysis must also include a test to assess how the planner behaves as the number of available agents (and corresponding skills) increases.

In the tests shown in the previous section, the number of agents per skill was 10 in the *Rescue Agents* scenario and 2 (per combination of skills) in the *Custom Balls Factory* scenario. The tests shown here, in Figure 5.13 (left diagram for *Rescue Agents* scenario and right diagram for *Custom Balls Factory* scenario), present the results for the same tests as in the previous section but with increasing number of agents per skill (or combination of skills).

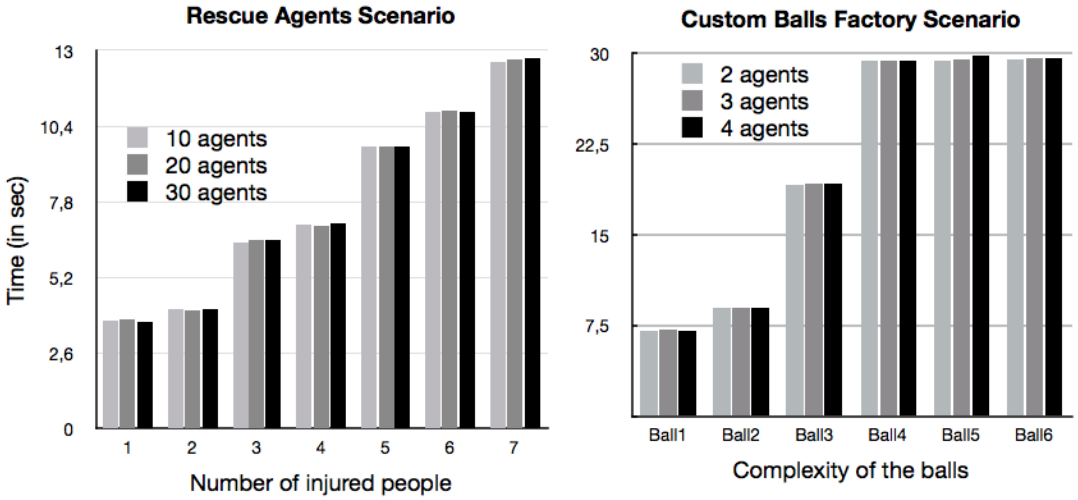


Figure 5.13: Evolution of performance of the Operators-graph-based Distributed Graphplan in both testing scenarios as skills distribution increase. The given number of agents is per skill.

As we can see, in both scenarios, there seems to be almost no variation in the overall performance of the planner as the number of agents per skill increases. The lack of variation is caused by the fact that, when the time comes to choose

an agent to which the partially-solved problem should be sent, even though the choosing agent now has a larger number of alternatives to consider, it chooses the appropriate agent randomly. Hence, the number of existing candidate agents is of no relevance to the performance of the planner.

In the previous section, we did not consider the time it took to generate the corresponding *semantic overlay network* for each scenario because it was equal for all the tests and it was too insignificant relatively to the overall planning time. However, now that the number of agents varies (and thus the *skill distribution factor*) the generation of the *semantic overlay network* is different for each test. Figure 5.14 depicts the time taken to generate the overlay network for each variation of the number of agents per skill, for both scenarios.

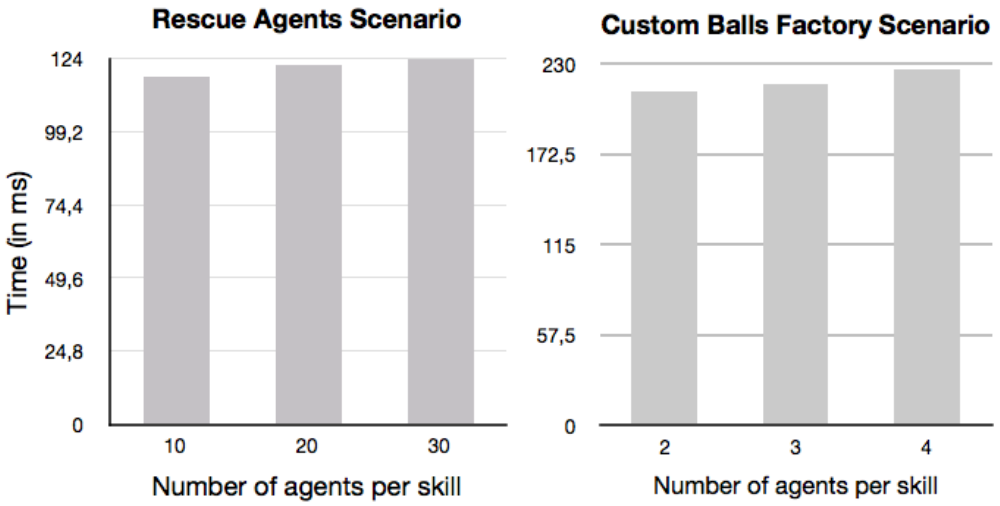


Figure 5.14: Comparison of time to generate the *semantic overlay network* as number of agents per skill varies.

The figure depicts a very slight variation of the time to generate the *semantic overlay network* as the number of agents per skill varies. Although the number of agents has significantly increased (twice or three times more), the time it takes to generate the semantic network is almost unchanged because, as the number of agents increases, the *skill distribution factor* decreases. As previously shown in sub-section 5.1.3, as the distribution factor decreases, the more likely it is to find each different skill in the network. Hence, the variation of the time that it

takes to complete the generation of the *semantic overlay network* in networks with more agents per skill is considerably smaller.

### 5.2.3 Open Conditions and Resolvers

Each agent only plays a small part in the overall problem solving process. When the agent realises that it can no longer contribute to the problem at hand, it must find a suitable agent that can potentially contribute to the unsolved sub-problems. As previously explained, the agent chooses one of the open conditions in the planning problem (propositions that remain unsatisfied in the current graph) and uses the semantic overlay network to determine which agents (and corresponding skills) can be used to further contribute to solve it. Once a list of candidate agents has been obtained, the agent must choose one to which the current problem will be forwarded.

Up to this point, the tests performed to evaluate our approach have made these decisions randomly. However, it is important to determine the influence a deeper or more sophisticated analysis of the open conditions and available resolver agents may have in the performance of the planning algorithm. One possible (and intuitive) approach is to quantify the contribution of each candidate agent by choosing the agent that can contribute to the largest number of open conditions in the current graph. We applied this heuristic to the planning algorithm and performed the same tests as in the previous sub-section, which results are shown in Figure 5.15.

We can see in the figure that the performance of the planner got worse as the number of agents per skill increased in both scenarios. However, the variation was smaller in the *Custom Balls Factory* scenario. This is related to the fact that each agent, when faced with the decision to choose the next agent to forward the planning graph, has to perform the same open conditions/skills analysis to a larger number of candidate agents. In the case of the *Custom Balls Factory* scenario, the number of candidate agents per combination of skill is much smaller than in the *Rescue Agents* scenario.

This leads us to conclude that a random approach is, in general, more suitable for choosing the next open condition/agent to proceed in the planning process.

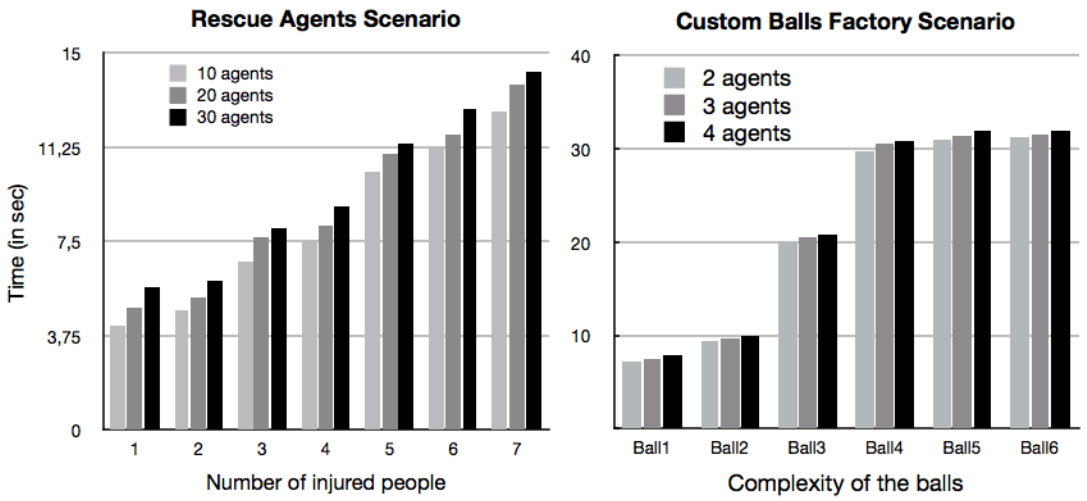


Figure 5.15: Evolution of performance of the Operators-graph-based Distributed Graphplan in both testing scenarios as skills distribution increase using a specific heuristic. The given number of agents is per skill.

Nevertheless, one cannot disregard the advantages of a more careful analysis such as the one depicted in this heuristic, just because, for these particular scenarios, the introduced overhead was too much to compensate for the gain in the performance of the planner. That is why we decided not to include this process in the decisions section (see section 4.3 in Chapter 4), since it may potentially come as an advantage in different problems. This is further discussed in the next chapter, in particular, in sub-section 6.2.3.

## 5.3 Evaluation Summary

Our approach to coordinate multiple heterogeneous agents in large unstructured networks is based on two complementary mechanisms: a distributed planning process that allows agents to decompose problems into sub-problems; and a discovery process that, through the dynamic generation and maintenance of a *semantic overlay network*, enables agents to find the agents with the skills they require to solve the unsolved sub-problems. In this chapter we have presented the tests that were performed to assess the quality of our approach.

First, we evaluated the discovery process by performing a set of tests aimed at analysing the performance of the algorithms used to generate a *semantic overlay network*. We compared our two algorithms, Priority-based Flooding (PbF – see sub-section 4.1.1) and Iterative Branching Depth-First Search (IBDFS – see sub-section 4.1.2), with alternative algorithms. The comparison (see section 5.1) was based on the worst case scenario of 1000 agents trying to generate a *semantic overlay network* at the same time. From this test, we concluded that the PbF algorithm, in spite of our initial hypothesis, was not a valid alternative, since its performance was not good enough to surpass other alternatives.

However, this first test has shown the potential of the IBDFS algorithm to perform better than alternative algorithms. In order to empirically show that this holds true, we continued testing this algorithm in different settings by changing several test parameters, such as *Time To Live* (see sub-section 5.1.1), number of neighbours (see sub-section 5.1.2), resource distribution (see sub-section 5.1.3) and disconnecting probabilities (see sub-section 5.1.4). In all of the tests, the IBDFS algorithm has consistently performed better than the alternatives.

We have also performed simulations in larger networks in order to determine the scalability of the algorithms (see sub-section 5.1.5). Once again, IBDFS was proven to be a far better alternative. This is due to the fact that IBDFS imposes less work on the agents by iteratively using an alternative neighbour only when it is required instead of making use of all possible alternatives right away. In order to explain this behaviour, we also showed (in sub-section 5.1.6) how the network load imposed by each of the algorithms has a direct link to their efficiency. The different tests performed on these algorithms allowed us to conclude that the IBDFS algorithm is an efficient, robust and scalable algorithm for searching and building *semantic overlays* in unstructured networks.

As referred above, our coordination approach encompasses the discovery process and the planning process. After showing that the IBDFS algorithm was an efficient approach to be used in the discovery process, we tested two planning algorithms, the distributed *Graphplan* algorithm (see sub-section 4.2.3) and the *operators-graph* based distributed *Graphplan* algorithm (see sub-section 4.2.4), in our overall distributed problem solving approach. The tests were done using examples of problems of the two scenarios described in Chapter 3.

The tests (see sub-section 5.2.1) indicate that the use of a goal-directed approach in the planning process is more suitable in larger and more complex problems. This is due to the fact that the simple generation of an *operators-graph* helps reduce the number of actions that are considered in the rest of the planning process, thus making the *operators-graph* based version scale a lot better than the regular version of the distributed *Graphplan* algorithm.

In the same sense that we wanted to test the performance of the algorithms for the generation of the *semantic overlay network* in different settings, we also tested the planning algorithms under different configurations of the same problems, *e.g.*, changing the number of agents and the distribution of skills (see sub-section 5.2.2), to evaluate their impact in the algorithm performance. These tests have shown that, as long as we use a random approach instead of a heuristic-based approach (see sub-section 5.2.3), the generation of the *semantic overlay network* and the performance of the algorithms is almost unaffected by changes in the number of agents and the distribution of skills in the network.

In conjunction with the conclusions of the previous sub-sections, this allows us to conclude that our approach of combining the IBDFS algorithm and the *operators-graph* based version of the distributed *Graphplan* algorithm is a step towards bringing efficiency, scalability and robustness to the distributed coordination of multiple agents for distributed problem solving in unstructured environments.

# Chapter 6

# Conclusions

Our research experience and the analysis of state-of-the-art work of the last few years in the field of multi-agent coordination and distributed problem solving has shown that focusing on structured environments presents undesirable limitations regarding robustness and scalability. Considering this, we set the goal for our research work: to create a robust, scalable and efficient coordination framework that enables agents to freely participate in distributed problem solving in totally unstructured agent societies.

For years, peer-to-peer (P2P) computing addressed similar challenges. However, P2P research focused mainly on the efficient management of the network, that is, finding ways for peers to effectively search and exchange information in a distributed network. However, each peer is treated, in P2P computing, as a simple reactive node, with little or no autonomy at all, thus ignoring the potential for developing collaborative environments. It was the distributed capabilities of P2P networks and the intelligence of autonomous agents that encouraged us to seek a realistic approach for developing a coordination framework for totally unstructured distributed environments in the combination of these two apparently separate fields.

In the end of this research, we have reasons to believe this thesis showed exactly how these two different areas of expertise can be combined to deliver a robust, efficient and scalable problem-solving framework for totally distributed unstructured environments.

## 6.1 Research Contributions

The analysis of related work allowed us to conclude that most multi-agent coordination approaches rely on a structure of some sort (from social rules to the existence of structuring elements in the society) to efficiently deploy distributed problem solving systems. But it has also showed that depending on such structure weakens the problem solving and coordination approaches especially in terms of scalability and robustness.

In order to overcome that dependency on structured elements, we analysed the work done in P2P computing and of some hybrid approaches that already made use of P2P computing to enhance multi-agent coordination systems. Although promising, current P2P research did not address all of the challenges associated with a coordination system that did not rely on a structured network. With those limitations in mind, we established the particular goals for our research work: develop a distributed planning algorithm capable of taking into account only partial knowledge of the domain in order to allow agents to partially contribute to specific problems; and design an efficient search algorithm that allows agents to search the unstructured environment for agents with necessary skills to further contribute to the problems.

Regarding the discovery process we developed two different search algorithms, the Priority-based Flooding (PbF) and the Iterative Branching Depth-First Search (IBDFS). These algorithms allow agents, not only, to search for the necessary capabilities to solve specific problems but also to collect useful information, by using network evolution techniques, to create and maintain a *semantic overlay network* that facilitates future searches. The evaluation of these algorithms concludes that, while the PbF algorithm is not a valid alternative, the IBDFS algorithm was proven to be the best algorithm to be used in the search of agents in the network and in the generation of the *semantic overlay network*.

Regarding the planning process, we developed two distributed versions of the *Graphplan* planning algorithm. The first version was a transformation of the *planning graph* generation into a distributed process, which included modifying the algorithm to account for the fact that each agent has only knowledge of its own capabilities and thus can only make partial contributions to that process.

However, the transformation of the *Graphplan* algorithm into a distributed process does not solve some of the limitations that this algorithm has, for example, the problem of "*distraction*", in which the algorithm considers all propositions of the initial state even if some of these are completely irrelevant to solve the problem. The second version of this algorithm applied "*means-ends analysis*" (focusing on the goals) to our distributed version of the *Graphplan* algorithm in order to diminish the effect of the "*distraction*" problem of the original algorithm. The tests performed on the distributed planning algorithms, which were done with problems from two different demonstration scenarios, show that the goal-directed version of the distributed *Graphplan* algorithm is efficient and scales a lot better than the regular distributed version of the algorithm.

It is the combination of these two different but complementary mechanisms that allows us to answer the research question initially presented in the introductory chapter: Yes, it is possible to create a robust, efficient and scalable system to coordinate the distributed problem solving activity of multiple heterogeneous agents in unstructured environments. The key to achieve that goal is the combination of a distributed version of an intelligent planning algorithm, as a general-purpose problem-solving and coordination tool, with also general-purpose peer-to-peer search and self-organisation algorithms as a robust and scalable means to discover the agents with the required capabilities to solve the problem.

## 6.2 Limitations and Future Work

The research described in this thesis presented compelling results regarding the cooperation of multiple agents in large unstructured environments. However, as most research, this is a work in progress and we aim to improve some of the aspects of the coordination system. In this section, we outline some of those aspects that need improvement, which will be the guidelines for future work.

### 6.2.1 Exploring Alternative Solutions

Our approach focuses on finding the agents with the necessary capabilities in the network, as efficiently as possible, and performing the necessary planning to find a solution to each problem. However, finding just the necessary capabilities to solve a problem may end up producing inefficient solution plans. This is particularly important in time-sensitive scenarios like *Rescue Agents* where it is essential that the entities in the environment act quickly in order to save the lives of the injured people.

Consider the following example: to rescue an injured person that is trapped inside some wreckage, our system would try to find a doctor and a fireman, which possess the necessary skills for the problem at hand. For this particular problem, this solution is, in fact, the optimal solution. However, imagine that there are, instead, several injured people and several doctors and firemen available. The system would still try to find only one doctor and one fireman (because that is enough to solve the problem) instead of delivering the optimal solution plan that would explore the possibility of using several doctors and firemen in parallel.

However, improving our system to address this limitation is not an easy task. For example, imagine that an agent has already produced a solution plan for a specific problem but that the plan could be improved by adding other entities that could work in parallel to reach a potentially faster execution. This situation raises several questions. How can an agent know that the plan that it currently holds, although enough to solve the problem, can be improved by adding new participants? The only way the agent has to know for sure is to continue the collaboration and continuously request the participation of new entities. But, if each agent is constantly assuming the solution plan can be improved by adding new participants, when does this process end? Maybe each agent can compare the resulting plans to determine if any improvements were actually made. If none were made, then the agent can assume the plan has reached an optimal state. Alternatively, each agent could perform the planning process with the goal of maximising a domain-dependent evaluation function that would, for example, value plans with more parallelism.

However, without those domain-dependent functions, these agents can only operate with the partial knowledge of the skills available in the network. It might be the case that an agent is not able of further improving a solution plan, thus considering it to be optimal, but the plan could, in fact, be improved, if different capabilities were available to the agent. In such distributed environments, there is never a guarantee that the solution plan is the best possible. Moreover, in order to come up with potentially better solution plans, the distributed problem solving process must continue to explore new possibilities, which may result in much longer planning processes. In fact, we have performed a few preliminary tests in which the agents were forced to search for a better solution plan (until no further improvements could be made) and, while the solution plans were in fact better (less steps in the execution phase), the planning phase took a lot more time than our original approach. So, although it can potentially lead to a longer solution plan, our approach has the advantage of providing a much faster planning process.

### 6.2.2 Acting on Behalf of Other Agents

Each agent in our system has only knowledge of its own skills. It is only after taking part in the self-organisation process of building the *semantic overlay network* that an agent becomes aware of the skills of other agents, especially, of those semantically related to it. In our approach, this information is only used to locate agents that have the necessary skills to complete the solution to a particular problem. Once the skill is located, the agent currently holding the partially-solved problem sends it to the agent holding the required skill so that it can contribute to the plan.

What if, instead, the first agent used that information directly in its planning process thus saving the time it takes to communicate with the other agent? This has the potential to speed up the planning process, but in doing so, the first agent is acting on behalf of the other agent in terms of commitment to participate in the solution plan. In other words, the first agent is assuming that, just because it has the necessary skills, the other agent will contribute to solve the given problem. This kind of assumption cannot be made because the first agent has no way of

knowing if the second agent can commit to play the required role in the solution plan or if there are any constraints preventing it from doing so, for example, having previously committed to participate in another plan that will clash with this one.

Planning with such strategy would only lead to solution plans that most likely would not be executed due to the fact that the participating agents cannot perform the required actions because local constraints, which were not considered at planning time, prevent them from committing to the actions on the solution plan. A possible alternative is for agents to engage in a negotiation process in which they exchange constraints. For example, the first agent, before adding the action to the plan, would ask permission to the second agent, to which it could reply, after checking current local constraints, whether it accepts it or not. These messages are potentially less "expensive" from the communications point-of-view because they are simple queries, compared to the size of the messages that are sent with partially-filled planning graphs. However, there may be more of them in quantity, which reduces the potential of this approach.

Agents cannot act on behalf of other agents unless they have their permission or they are aware of their constraints. In both cases, heavy communication may be required. However, once an agent is aware of other agents constraints, it would no longer have to ask for them again (assuming these do not change over time and that the agent is in possession of all the constraints and not just a subset). This is not a safe assumption to make, especially in highly dynamic environments, but it may be of relevance for problems in which a continuous collaboration between two or more agents is required. For some agents, contributing to a solution plan only requires a small participation, that is, the number of times its actions appear in the final solution plan is quite small. However, in scenarios as the *Rescue Agents*, most participating agents have a more determining role in the solution plan, as a doctor having to provide assistance to 6 injured people located in different areas of a city. This problem, which requires the participation of a doctor and an ambulance driver, will continuously be sent back and forth between the two agents representing these two entities so that each can add its actions to the solution plan. A lot of communication can be saved if one of the agents simply performs the planning once all local constraints and necessary actions are known.

For example, consider that the first agent receiving the problem is the medic agent, which after processing it, determines that it can provide assistance to one of the injured people, but for that it requires the participation of an ambulance that can take him there. So, it contacts the ambulance agent by sending it the partially-solved problem. At this point, the ambulance agent already has the necessary knowledge to perform the entire planning process that would involve providing assistance to the remaining injured people. However, as explained before, it cannot commit to the plan on behalf of the medic agent unless it has its permission or it is aware of its constraints. But, if the medic agent, when sending the partially-solved problem to the ambulance agent, would also include its local constraints (as an implicit authorisation to act on its behalf), then the ambulance agent could build the entire solution plan, thus saving a lot of messages in the process. We performed some preliminary tests and, in fact, the problems of the *Rescue Agents* scenario were solved in less time than originally, whereas the problems in the *Custom Balls Factory* scenario had little or no improvement at all. Nevertheless, this approach, which is based on a potentially unsafe assumption that agents can commit to the plans on behalf of other agents (as long as they know their constraints), needs to be further analysed.

An alternative approach could be based on abstract commitments at the planning stage that would only be realised at the execution stage. That is, an agent building the solution plan could include abstract commitments with the skills that it found on the network. These commitments are abstract in the sense that no actual agent has committed to them. They are only associated to a skill found in the network. Then, at the execution stage, agents with the necessary skills and that have no local constraints that would unable them to commit to those abstract slots in the plan, would be contacted to perform those parts of the plan. We have worked on similar approaches before (Botelho *et al.*, 2008b) but further research is necessary to consider dynamic unstructured environments. This is something that we plan to do in the future.

### 6.2.3 Choosing Appropriate Resolvers Based on Context

Each agent in this distributed problem solving process, after determining how its own skills can be used to partially contribute to the solution, must find a suitable agent that can potentially contribute to the unsolved parts of the problem. In most situations, this includes having to choose a particular agent from a long list of candidates, which may influence the performance of the system. This decision was first mentioned in section 4.2.3, in which we stated that the agent was chosen randomly, and later on discussed an alternative approach relying on an heuristic that would quantify the potential contribution of each candidate agent (see section 5.2.3).

The use of a random approach in choosing the adequate agent to contribute to the solution was justified simply by the fact that it was faster than choosing the agent that can solve more open conditions, in all performed tests. The random approach is faster because the overhead introduced by the heuristic approach was too much to compensate the improvement brought by its application. However, in more complex environments, such as the ones in which agents commit and act based on costs and rewards, a random approach can be very inefficient, leading to very costly solution plans. In such cases, the challenge revolves around identifying the information that should be used to select the appropriate agent.

The quantifiable contributions and the costs and rewards associated with the commitment of chosen agents are very important to *evaluate* potential candidates, but other different sources of information can also be very useful. Information such as the agent's general availability, workload, location and past average performance are just a few examples of contextual data that, in combination with other relevant data, can be used to narrow down the list of potential candidates.

Combining all of these considerations into a unified context-aware system is quite a challenge, but it is one in which we have already presented some promising research work (Botelho *et al.*, 2008b) (Costa *et al.*, 2008). Even though this was not initially one of the goals of our research, we intend to evaluate how a context-aware based process can be used in such distributed unstructured environments to improve the process of choosing the appropriate resolvers for partial contributions in distributed problem solving.

## 6.3 Final Words

This thesis is an essential element to the work we have been developing in the last few years in the field of intelligent agents and multi-agent systems. We tried to fully substantiate, describe and evaluate our approach for cooperative distributed problem solving in unstructured environments.

Each agent in our approach uses a goal-directed version of the distributed *Graphplan* algorithm to make partial contributions to the generation of a solution plan to a given problem. Agents with the necessary skills to contribute to unsolved parts are easily and efficiently discovered resorting to a semantic overlay network dynamically built and maintained by the discovery process itself. The evaluation of our approach, based on two very distinct scenarios, has consistently shown that it is robust, scalable and efficient and that, although with some room for improvement (as explained above), it can be generically applied to large and complex distributed problem solving environments.

These promising results encourage us to proceed with this research and overcome its limitations. Thus, this thesis does not mark the end of our research in this particular subject. On the contrary, it is rather a milestone in our vision to build real world intelligent agent societies, in which autonomous agents can seamlessly cooperate by combining their skills to build a collective intelligence capable of solving complex problems.

# References


Abdallah, S. & Lesser, V. (2004). Organization-based cooperative coalition formation. In *Proceedings of the IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology*, 162–168. 22, 37

Adjiman, P., Chatalic, P., Goasdoue, F., Rousset, M. & Simon, L. (2006). Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *Journal of Artificial Intelligence Research*, **25**, 5–6. 35

Amigoni, F., Gatti, N., Pinciroli, C., Roveri, M. & e Informazione, D. (2005). What planner for ambient intelligence applications? *IEEE Transactions on Systems, Man and Cybernetics, Part A*, **35**, 7–21. 21

Anderson, C., Smith, D.E. & Weld, D.S. (1998). Conditional effects in graphplan. In *Proceedings of the 4th Int. Conf. on AI Planning Systems*. 18, 37

Arabshian, K. & Schulzrinne, H. (2007). An ontology-based hierarchical peer-to-peer global service discovery system. *Journal of Ubiquitous Computing and Intelligence*, **1**, 133–144. 33

Arpinar, I.B., Zhang, R., Aleman-Meza, B. & Maduko, A. (2005). Ontology-driven web services composition platform. *Information Systems and E-Business Management*, **3**, 175–199. 34

Babaoglu, O., Meling, H. & Montresor, A. (2002). Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the 22nd Int. Conf. on Distributed Systems*, 15–22. 35, 36

Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R. & Stoica, I. (2003). Looking up data in p2p systems. *Communications ACM*, **46**, 43–48. 32

Banaei-kashani, F., Chen, C. & Shahabi, C. (2004). Wspds: Web services peer-to-peer discovery service. In *Proceedings of the Int. Conf. on Internet Computing*, 733–743, Citeseer. 33

Ben-Ami, D. & Shehory, O. (2005). A comparative evaluation of agent location mechanisms in large scale mas. In *Proceedings of the 4th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 339–346, ACM New York, NY, USA. 37

Benatallah, B., Dumas, M., Sheng, Q. & Ngu, A. (2002). Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th Int. Conf. on Data Engineering*, 297–308. 34

Bianchini, D., De Antonellis, V., Melchiori, M. & Salvi, D. (2006). Peer-to-peer semantic-based web service discovery: state of the art. Tech. rep., Dipartimento di Elettronica per l'Automazione Università di Brescia. 29, 31

Blankenburg, B., Botelho, L., Calhau, F., Fernández, A., Klusch, M. & Ossowski, S. (2008). *Intelligent Service Coordination in the Semantic Web*, chap. Service Composition, 235–262. Birkhauser. 4

Blum, A. & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, **90**, 281–300. 17, 18, 37, 71, 73

Bonnet-Torres, O. & Tessier, C. (2005). From team plan to individual plans: a petri net-based approach. In *Proceedings of the 4th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 797–804, ACM New York, NY, USA. 21

Botelho, L., Mendes, H., Figueiredo, P. & Marinheiro, R. (2003). Send fredo off to do this, send fredo off to do that. In *Proceedings of the International Workshop on Collaborative Information Agents*. 3

Botelho, L., Fernández, A., Fries, B., Klusch, M., Pereira, L., Santos, T., Pais, P. & Vasirani, M. (2008a). *Intelligent Service Coordination in the Semantic Web*, chap. Service Discovery, 205–233. Birkhauser. 4

Botelho, L., Lopes, A., Moller, T. & Schuldt, H. (2008b). *Intelligent Service Coordination in the Semantic Web*, chap. Semantic Web Service Execution, 263–287. Birkhauser. 4, 129, 130

Costa, P., Gonçalves, B. & Botelho, L. (2008). *Intelligent Service Coordination in the Semantic Web*, chap. Context-Awareness System, 289–308. Birkhauser. 4, 28, 130

Cox, J. & Durfee, E. (2005). An efficient algorithm for multiagent plan coordination. In *Proceedings of the 4th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 828–835, ACM New York, NY, USA. 26

Cox, M., Elahi, M. & Cleereman, K. (2003). A distributed planning approach using multiagent goal transformations. In *Proceedings of the 14th Midwest Artificial Intelligence and Cognitive Science Conference*, 18–23. 24

Crespo, A. & Garcia-Molina, H. (2002). Routing indices for peer-to-peer systems. In *Proceedings of the 22nd Int. Conf. on Distributed Computing Systems*, 23, IEEE Computer Society, Washington, DC, USA. 30, 31, 36, 38

Crespo, A. & Garcia-Molina, H. (2005). Semantic overlay networks for p2p systems. In *Proceedings of the 3rd Int. Workshop on Agents and Peer-to-Peer Computing*, 1, Springer, New York, NY, USA. 33, 37, 64

Dale, J. & Ceccaroni, L. (2002). Pizza and a movie: A case study in advanced web services. In *Proceedings of the Agentcities: Challenges in Open Agent Environments Workshop of AAMAS*. 3

Dasgupta, P. (2005). Improving peer-to-peer resource discovery using mobile agent based referrals. *Lecture notes in computer science*, **Volume 2872/2005**, 186–197. 36

Davis, R. & Smith, R.G. (1983). Negotiation as a metaphor for distributed problem solving. *Artificial intelligence*, **20**, 63–100. 22

de Weerdt, M., ter Mors, A. & Witteveen, C. (2005). Multi-agent planning: An introduction to planning and coordination. In *Handouts of the European Agent Summer School*, 1–32. 22, 26

de Weerdt, M., Zhang, Y. & Klos, T. (2007). Distributed task allocation in social networks. In *Proceedings of the 6th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, ACM New York, NY, USA. 22, 37

Desjardins, M., Durfee, E., Ortiz, C. & Wolverton, M. (1999). A survey of research in distributed, continual planning. *Artificial Intelligence Magazine*, **20**, 13–22. 28

Dimakopoulos, V. & Pitoura, E. (2003). A peer-to-peer approach to resource discovery in multi-agent systems. *Lecture notes in computer science*, **2782**, 62–77. 36

Durfee, E. (1999). Distributed problem solving and planning. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*, 121–164. 1, 22

Durfee, E. & Lesser, V. (1989). Negotiating task decomposition and allocation using partial global planning. *Distributed artificial intelligence*, 229. 19

Ephrati, E. & Rosenschein, J. (1994). Divide and conquer in multi-agent planning. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 375–375. 25

Ephrati, E., Pollack, M. & Ur, S. (1995). Deriving multi-agent coordination through filtering strategies. In *Proceedings of the Int. Joint Conf. of Artificial Intelligence*, vol. 14, 679–687. 23

Ermolayev, V., Keberle, N., Plaksin, S. & Kononenko, O. (2004). Towards a framework for agent-enabled semantic web service composition. *International Journal of Web Services Research*, **1**, 63–87. 34

Erol, K. (1996). *Hierarchical task network planning: formalization, analysis, and implementation*. Ph.D. thesis, University of Maryland, College Park, MD, USA. 21

Erol, K., Hendler, J. & Nau, D. (1995). Htn planning: Complexity and expressivity. In *Proceedings of the National Conference on Artificial Intelligence*, 1123–1123. 21

Fikes, R. & Nilsson, N. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, **2**, 189–208. 13, 36, 71

Fletcher, G.H.L. & Sheth, H.A. (2004). Unstructured peer-to-peer networks: Topological properties and search performance. In *Proceedings of the Agents and Peer-to-Peer Computing Workshop*, 14–27, Springer. 64

Fung, R. & Chen, T. (2005). A multiagent supply chain planning and coordination architecture. *International Journal of Advanced Manufacturing Technology*, **25**, 811–819. 22, 37

Gaston, M. & Desjardins, M. (2005). Agent-organized networks for dynamic team formation. In *Proceedings of the 4th Int. Joint Conf. on Autonomous agents and multiagent systemsnternational joint conference on Autonomous Agents and Multiagent Systems*, 230–237, ACM New York, NY, USA. 22, 37

Georgeff, M.P. (1988). Communication and interaction in multi-agent planning. *Distributed artificial intelligence*, 200–204. 25

Ghallab, M., Nau, D. & Traverso, P. (2004). *Automated Planning: theory and practice*. Morgan Kaufmann Publishers. 14, 16, 92

Haase, P., Agarwal, S. & Sure, Y. (2004). Service-oriented semantic peer-to-peer systems. In *Proceedings of the Workshop on Web Information Systems*, 46, Springer-Verlag New York Inc. 34

Helin, H., Klusch, M., Lopes, A., Fernández, A., Schumacher, M., Schuldt, H., Bergenti, F. & Kinnunen, A. (2005). Cascom: Context-aware service co-ordination in mobile p2p environments. *Lecture Notes on Computer Science*, **Volume 3550/2005**, 242–243. 3

Jamali, N. & Zhao, X. (2005a). Hierarchical resource usage coordination for large-scale multi-agent systems. *Lecture Notes on Artificial Intelligence: Massively Multi-agent Systems I*, **3446**, 40–54. 22, 37

Jamali, N. & Zhao, X. (2005b). A scalable approach to multi-agent resource acquisition and control. In *Proceedings of the 4th int. joint conf. on Autonomous agents and multiagent systems on Autonomous agents and multiagent systems*, 868–875, ACM New York, NY, USA. 37

Jennings, N., Sycara, K. & Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, **1**, 7–38. 22

Jin, H., Wu, H., Li, Y. & Chen, H. (2005). Semantic overly-driven web services discovery. In *Proceedings of the 1st Int. Conf. on Semantics, Knowledge and Grid*, 76–87. 33

Kalogeraki, V., Gunopulos, D. & Zeinalipour-Yazti, D. (2002). A local search mechanism for peer-to-peer networks. In *Proceedings of the 11th Int. Conf. on Information and Knowledge Management*, 300–307, ACM, New York, NY, USA. 31, 36, 38

Kambhampati, S., Parker, E. & Lambrecht, E. (1997). Understanding and extending graphplan. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning*, 260–272, Springer-Verlag London, UK. 18, 37, 84

Küngas, P. & Matskin, M. (2006). Semantic web service composition through a p2p-based multi-agent environment. *Lecture Notes on Computer Science*, **4118**, 106–119. 5, 6, 34

Li, T.Y., Zhao, Z.G. & You, S.Z. (2003). A-peer: An agent platform integrating peer-to-peer network. In *Proceedings of the 3rd Int. Symposium on Cluster Computing and the Grid*, 614, IEEE Computer Society, Washington, DC, USA. 34

Liang, J., Kumar, R. & Ross, K.W. (2006). The fasttrack overlay: a measurement study. *Computer Networks*, **50**, 842–858. 32, 38

Liu, J. & Zhuge, H. (2005). A semantic-link-based infrastructure for web service discovery in p2p networks. In *Proceedings of the International World Wide Web Conference*, 940–941, ACM New York, NY, USA. 33

Lopes, A. & Botelho, L. (2007). Task decomposition and delegation algorithms for coordinating unstructured multi agent systems. In *Proceedings of the 1st Int. Conf. on Complex, Intelligent and Software Intensive Systems*, 209–214. 27

Lv, Q., Cao, P., Cohen, E., Li, K. & Shenker, S. (2002). Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th Int. Conf. on Supercomputing*, 84–95, ACM, New York, NY, USA. 31, 38

McDermott, D. (2000). The 1998 ai planning systems competition. *AI Magazine*, **21**. 41

Ogston, E. & Vassiliadis, S. (2002). A peer-to-peer agent auction. In *Proceedings of the 1st Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 151–159, ACM New York, NY, USA. 37

Ratnasamy, S., Francis, P., Handley, M., Karp, R. & Schenker, S. (2001). A scalable content-addressable network. In *Proceedings of the 2001 SIGCOMM conference*, vol. 31-4, 161–172, ACM New York, NY, USA. 32, 38

Ratsimor, O., Chakraborty, D., Joshi, A., Finin, T. & Yesha, Y. (2004). Service discovery in agent-based pervasive computing environments. *Mobile Networks and Applications*, **9**, 679–692. 30

Romeikat, R. & Bauer, B. (2007). Towards semantically-enhanced distributed service discovery. In *Proceedings of the 2nd Int. Conf. on Internet and Web Applications and Services*, IEEE Computer Society Washington, DC, USA. 34

Rowstron, A.I.T. & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM Int. Conf. on Distributed Systems Platforms*, 329–350, Springer-Verlag, London, UK. 32, 38

Sacerdoti, E. & Center, A. (1975). The nonlinear nature of plans. In *Proceedings of the 4th Int. Joint Conf. on Artificial Intelligence*, 206, Morgan Kaufmann. 15, 36

Sapkota, B., Vasiliu, L., Toma, I., Roman, D. & Bussler, C. (2005). Peer-to-peer technology usage in web service discovery and matchmaking. *Lecture notes in computer science*, **3806**, 418–425. 33

Scerri, P., Owens, S., Yu, B. & Sycara, K. (2007). A decentralized approach to space deconfliction. In *Proceedings of the 10th Int. Conf. on Information Fusion*, Quebec, Canada. 26

Schumacher, M., de Oliveira e Sousa, A., Constantinescu, I., van Pelt, T. & Faltings, B. (2008). *Intelligent Service Coordination in the Semantic Web*, chap. Distributed Directories of Web Services, 181–203. Birkhauser. 4

Shoham, Y. & Tennenholtz, M. (1992). On the synthesis of useful social laws for artificial agent societies. In *Proceedings of the National Conference on Artificial Intelligence*, 276–276. 23, 37

Smith, D. & Peot, M. (1996). Suspending recursion in causal-link planning. In *Proceedings of the 3rd Int. Conf. on Articial Intelligence Planning Systems*. 84

Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 149–160, ACM, New York, NY, USA. 32, 38

Sugawara, T., Kurihara, S., Fukuda, K., Hirotsu, T., Aoyagi, S. & Takada, T. (2004). Reusing coordination and negotiation strategies in multi-agent systems for ubiquitous network environment. In *Proceedings of the 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 496–503, IEEE Computer Society Washington, DC, USA. 27

Ter Mors, A., Valk, J., Witteveen, C., Arabnia, H.R. & Mun, Y. (2004). Coordinating autonomous planners. In *Proceedings of the Int. Conf. on Artificial Intelligence*, 795–801. 37

Tews, A. & Wyeth, G. (2000). Maps: A system for multi-agent coordination. *Advanced Robotics*, **14**, 37–50. 23, 27

Toma, I., Sapkota, B., Secuila, J., Gomez, J.M., Roman, D. & Bussler, C. (2005). P2p discovery mechanisms for web service execution environment. In *Proceedings of the 2nd WSMO Implementation Workshop*. 33

Tsamardinos, I., Pollack, M. & Horty, J. (2000). Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *Proceedings of the 5th Int. Conf. on Artificial Intelligence Planning and Scheduling*, 264–272. 26

Tsoumakos, D. & Roussopoulos, N. (2003). Adaptive probabilistic search for peer-to-peer networks. In *Proceedings of the 3rd Int. Conf. on Peer-to-Peer Computing*, 102, IEEE Computer Society, Washington, DC, USA. 31, 38

Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S. & Miller, J. (2005). Meteor-s wsdi: a scalable infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management, Special Issue on Universal Global Integration*, **6**, 17–39. 34

Vouros, G. (2007). Information searching and sharing in large-scale dynamic networks. In *Proceedings of the 6th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, ACM New York, NY, USA. 31

Walsh, W. (1999). A market protocol for decentralized task allocation and scheduling with hierarchical dependencies. In *Proceedings of the 3rd Int. Conf. on Multi-Agent Systems*, 325–332. 22, 37

Walsh, W., Wellman, M. & Ygge, F. (2000). Combinatorial auctions for supply chain formation. In *Proceedings of the 2nd ACM Conf. on Electronic Commerce*, 260–269, ACM New York, NY, USA. 22

Weld, D. (1999). Recent advances in ai planning. *Artificial Intelligence Magazine*, **20**, 93–123. 18, 75

Weld, D.S., Anderson, C.R. & Smith, D.E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National/Tenth Conf. on Artificial intelligence/innovative Applications of Artificial intelligence*, vol. 98, 897–904. 18, 37

Wellman, M. (1993). A market-oriented programming environment and its application to distributed multi commodity flow problems. *Journal of Artificial Intelligence Research*, **1**, 1–23. 22

Wellman, M. (1996). Market-oriented programming: Some early lessons. *Market-based control: a paradigm for distributed resource allocation*, 74–95. 22

Wellman, M., Walsh, W., Wurman, P. & MacKie-Mason, J. (2001). Auction protocols for decentralized scheduling. *Games and Economic Behavior*, **35**, 271–303. 22

Willmott, S., Dale, J., Burg, B., Charlton, P. & O'Brien, P. (2001). Agentcities: A worldwide open agent network. In *Agentlink Newsletter*, vol. 8, 13–15. 2

Willmott, S., Pujol, J.M. & Cortés, U. (2005). On exploiting agent technology in the design of peer-to-peer applications. *Lecture Notes on Computer Science*, **3601/2005**, 98–107. 6

Yang, B. & Garcia-Molina, H. (2002). Efficient search in peer-to-peer networks. In *Proceedings of the Int. Conf. on Distributed Computing Systems*. 30, 31, 36, 38

Yu, S., Liu, J. & Le, J. (2004). Decentralized web service organization combining semantic web and peer to peer computing. *Lecture notes in computer science*, **3250**, 116–127. 33

Zhao, B.Y., Kubiatowicz, J.D. & Joseph, A.D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. rep., University of Berkeley, Berkeley, CA, USA. 32, 38

# Appendix A

# Written Papers

**Published**:

- Lopes, A. L., Botelho, L. M. (2010). **Coordination in Open and Unstructured Intelligent Agent Societies: using distributed planners on top of a semantic overlay network**. In Proceedings of the *Second International Conference on Agents and Artificial Intelligence*.
- Lopes, A. L., Botelho, L. M. (2008). **Improving Multi-Agent Based Resource Coordination in Peer-to-Peer Networks**. *Journal of Networks, 3:(2)*, 38-47.
- Lopes, A. L., Botelho, L. M. (2008). **Efficient Algorithms for Agent-Based Semantic Resource Discovery**. In Proceedings of the *Seventh International Workshop on Agents and Peer to Peer Computing*, 1-15.
- Lopes, A. L., Botelho, L. M. (2007). **Task Decomposition and Delegation Algorithms for Coordinating Unstructured Multi Agent Systems**. In Proceedings of the *International Conference on Complex, Intelligent and Software Intensive Systems*, 209-214.

**Other**:

- Lopes, A. L., Botelho, L. M. (2007). **Task Decomposition and Delegation Algorithms for Coordinating Unstructured Multi Agent Systems**. In Proceedings of the *Doctoral Mentoring Program of the Sixth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, 26-27. **Mentor**: Professor Edmund Durfee - University of Michigan.

# Appendix B

# Testing Scenarios Listings

Listing B.1: *Ambulance driver*'s actions in the *Rescue Agents* domain

```
(:action ambulance_move
  :parameters (?a - ambulance ?loc1 ?loc2 - location)
  :precondition (and
    (ambulance_at ?a ?loc1)
    (adjacent ?loc1 ?loc2)
    (road ?loc2))
  :effect (and
    (ambulance_at ?a ?loc2)
    (not (ambulance_at ?a ?loc1)))
)
```

Listing B.2: *Fireman*'s actions in the *Rescue Agents* domain

```
(:action fireman_move
  :parameters (?fm - fireman ?loc1 ?loc2 - location)
  :precondition (and
```

```
    (fireman_at ?fm ?loc1)
    (adjacent ?loc1 ?loc2)
    (road ?loc2))
  :effect (and
    (fireman_at ?fm ?loc2)
    (not (fireman_at ?fm ?loc1)))
)

(:action fireman_putout_fire
  :parameters (?fm - fireman ?loc1 ?loc2 - location)
  :precondition (and
    (fireman_at ?fm ?loc1)
    (adjacent ?loc1 ?loc2)
    (fire ?loc2))
  :effect (and
    (road ?loc2)
    (not (fire ?loc2)))
)
```

Listing B.3: *Paramedic*'s actions in the *Rescue Agents* domain

```
(:action paramedic_triage
  :parameters (?pm - paramedic ?p - person ?loc - location)
  :precondition (and
    (paramedic_at ?pm ?loc)
    (person_at ?p ?loc)
    (person_injured ?p))
  :effect (and
    (person_triaged ?p))
```

```
)

(:action paramedic_load_in_ambulance
  :parameters (?pm - paramedic ?a - ambulance ?loc - location)
  :precondition (and
    (paramedic_at ?pm ?loc)
    (ambulance_at ?a ?loc))
  :effect (and
    (paramedic_at_ambulance ?pm ?a)
    (not (paramedic_at ?pm ?loc)))
)

(:action paramedic_unload_from_ambulance
  :parameters (?pm - paramedic ?a - ambulance ?loc - location)
  :precondition (and
    (paramedic_at_ambulance ?pm ?a)
    (ambulance_at ?a ?loc))
  :effect (and
    (paramedic_at ?pm ?loc)
    (not (paramedic_at_ambulance ?pm ?a)))
)
```

Listing B.4: PDDL description of an example problem in the *Rescue Agents* domain

```
(define (problem TestingScenarios)
  (:domain RescueAgents)

  (:objects
    l00 l01 l02 l03 l04 l05 l06 l07 l08 l09
```

```
    l10 l11 l12 l13 l14 l15 l16 l17 l18 l19
    l20 l21 l22 l23 l24 l25 l26 l27 l28 l29
    l30 l31 l32 l33 l34 l35 l36 l37 l38 l39
    l40 l41 l42 l43 l44 l45 l46 l47 l48 l49
    l50 l51 l52 l53 l54 l55 l56 l57 l58 l59
    l60 l61 l62 l63 l64 l65 l66 l67 l68 l69
    l70 l71 l72 l73 l74 l75 l76 l77 l78 l79
    l80 l81 l82 l83 l84 l85 l86 l87 l88 l89
    l90 l91 l92 l93 l94 l95 l96 l97 l98 l99 - location
    p1 - person )

  (:init
    (other l00)
    (adjacent l00 l10)
    (adjacent l00 l01)
    (other l01)
    (adjacent l01 l11)
    (adjacent l01 l00)
    (adjacent l01 l02)
    (road l02)
    (adjacent l02 l12)
    (adjacent l02 l01)
    (adjacent l02 l03)
    (...) ; This listing has been truncated to make it more legible

    ; Injured person details
    (person_injured p1)
    (person_at p1 l84)
  )
```

```
  (:goal
    (and
      (person_triaged p1) ) )
)
```

Listing B.5: Initial state description for the *ambulance* in the example problem in the *Rescue Agents* domain

```
(define (problem ambulance1_state)
    (:domain RescueAgents)

    (:objects
      l20 - location
      amb1 - ambulance
    )

    (:init
      (ambulance_at amb1 l20)
    )

    (:goal
      (and ))
)
```

Listing B.6: Initial state description for the *fireman* in the example problem in the *Rescue Agents* domain

```
(define (problem fireman1_state)
    (:domain RescueAgents)

    (:objects
      l27 - location
```

```
      fman1 - fireman
    )

    (:init
      (fireman_at fman1 l27)
    )

    (:goal
      (and ))
)
```

Listing B.7: Initial state description for the *paramedic* in the example problem in the *Rescue Agents* domain

```
(define (problem paramedic1_state)
    (:domain RescueAgents)

    (:objects
      l20 - location
      pmedic1 - paramedic
    )

    (:init
      (paramedic_at pmedic1 l20)
    )

    (:goal
      (and ))
)
```

Listing B.8: *Painter*'s actions in the *Custom Ball Factory* domain

```
(:action grab_ball
 :parameters (?p - painter ?b - ball)
 :precondition (and
          (painter_free ?p)
          (ball_free ?b))
 :effect (and
          (painter_has ?p ?b)
          (not (painter_free ?p))
          (not (ball_free ?b)))
)

(:action paint
 :parameters (?p - painter ?b - ball ?a - area ?c - colour)
 :precondition (and
          (painter_has ?p ?b)
          (can_colour ?p ?c)
          (can_paint ?p ?a))
 :effect (and
          (painted ?b ?a ?c))
)

(:action drop_ball
 :parameters (?p - painter ?b - ball)
 :precondition (and
          (painter_has ?p ?b))
 :effect (and
          (not (painter_has ?p ?b))
          (painter_free ?p)
          (ball_free ?b))
```

```
)
```

Listing B.9: *Stripes painter*'s actions in the *Custom Ball Factory* domain

```
(:action grab_ball
 :parameters (?sp - spainter ?b - ball)
 :precondition (and
          (spainter_free ?sp)
          (ball_free ?b))
 :effect (and
          (spainter_has ?sp ?b)
          (not (spainter_free ?sp))
          (not (ball_free ?b)))
)

(:action paint_stripes
 :parameters (?sp - spainter ?b - ball ?a - area ?pc ?nc - colour)
 :precondition (and
          (spainter_has ?sp ?b)
          (can_colour ?sp ?nc)
          (can_stripe ?sp ?a)
          (painted ?b ?a ?pc))
 :effect (and
          (striped ?b ?a ?pc ?nc)
          (not (painted ?b ?a ?pc)))
)

(:action drop_ball
 :parameters (?sp - spainter ?b - ball)
 :precondition (and
          (spainter_has ?sp ?b))
```

```
 :effect (and
          (not (spainter_has ?sp ?b))
          (spainter_free ?sp)
          (ball_free ?b))
)
```

Listing B.10: *Inflater*'s actions in the *Custom Ball Factory* domain

```
(:action grab_ball
 :parameters (?i - inflater ?b - ball)
 :precondition (and
          (inflater_free ?i)
          (ball_free ?b))
 :effect (and
          (inflater_has ?i ?b)
          (not (inflater_free ?i))
          (not (ball_free ?b)))
)

(:action inflate_ball
 :parameters (?i - inflater ?b - ball)
 :precondition (and
          (assembled ?b)
          (inflater_has ?i ?b))
 :effect (and
          (inflated ?b))
)

(:action drop_ball
 :parameters (?i - inflater ?b - ball)
 :precondition (and
```

```
            (inflater_has ?i ?b))
 :effect (and
            (not (inflater_has ?i ?b))
            (inflater_free ?i)
            (ball_free ?b))
)
```

Listing B.11: *Assembler*'s actions in the *Custom Ball Factory* domain

```
(:action grab_ball
 :parameters (?a - assembler ?b - ball)
 :precondition (and
            (assembler_free ?a)
            (ball_free ?b))
 :effect (and
            (assembler_has ?a ?b)
            (not (assembler_free ?a))
            (not (ball_free ?b)))
)

(:action assemble_ball
 :parameters (?a - assembler ?b - ball)
 :precondition (and
            (assembler_has ?a ?b))
 :effect (and
            (assembled ?b))
)

(:action drop_ball
 :parameters (?a - assembler ?b - ball)
 :precondition (and
```

```
        (assembler_has ?a ?b))
 :effect (and
        (not (assembler_has ?a ?b))
        (assembler_free ?a)
        (ball_free ?b))
)
```

Listing B.12: Initial state description for the *trgb_painter* agent in the example problem in the *Custom Ball Factory* domain

```
(define (problem trgb_painter_state)
    (:domain CustomBallFactory)

   (:objects
      red green blue - colour
      trgb_painter - painter
      tl tr - area       ; tl = top left section ; tr = top right
          section
    )

    (:init
      (painter_free trgb_painter)
      (can_colour trgb_painter red)
      (can_colour trgb_painter green)
      (can_colour trgb_painter blue)
      (can_paint trgb_painter tl)
      (can_paint trgb_painter tr)
    )

    (:goal
      (and  ))
```

```
)
```

Listing B.13: Initial state description for the *bb_painter* agent in the example problem in the *Custom Ball Factory* domain

```
(define (problem bb_painter_state)
    (:domain CustomBallFactory)

   (:objects
      blue - colour
      bb_painter - painter
      bl br - area       ; bl = bottom left section ; br = bottom
          right section
    )

    (:init
      (painter_free bb_painter)
      (can_colour bb_painter blue)
      (can_paint bb_painter bl)
      (can_paint bb_painter br)
    )

    (:goal
      (and  ))
)
```

Listing B.14: Initial state description for the *bs_painter* agent in the example problem in the *Custom Ball Factory* domain

```
(define (problem bs_painter_state)
    (:domain CustomBallFactory)
```

```
   (:objects
      blue - colour
      bs_painter - spainter
      tl tr bl br - area
    )

    (:init
      (spainter_free bs_painter)
      (can_colour bs_painter blue)
      (can_stripe bs_painter tl)
      (can_stripe bs_painter tr)
      (can_stripe bs_painter bl)
      (can_stripe bs_painter br)
    )

    (:goal
      (and  ))
)
```

Listing B.15: Initial state description for the *gs_painter* agent in the example problem in the *Custom Ball Factory* domain

```
(define (problem gs_painter_state)
    (:domain CustomBallFactory)

   (:objects
      green - colour
      gs_painter - spainter
      tl tr bl br - area
    )
```

```
    (:init
      (spainter_free gs_painter)
      (can_colour gs_painter green)
      (can_stripe gs_painter tl)
      (can_stripe gs_painter tr)
      (can_stripe gs_painter bl)
      (can_stripe gs_painter br)
    )

    (:goal
      (and  ))
)
```

Listing B.16: Initial state description for the *assemb* agent in the example problem in the *Custom Ball Factory* domain

```
(define (problem assemb_state)
    (:domain CustomBallFactory)

   (:objects
      assemb - assembler
    )

    (:init
  (assembler_free assemb))

    (:goal
      (and  ))
)
```

Listing B.17: Initial state description for the *infla* agent in the example problem in the

*Custom Ball Factory* domain

```
(define (problem infla_state)
    (:domain CustomBallFactory)

   (:objects
      infla - inflater
    )

    (:init
  (inflater_free infla))

    (:goal
      (and  ))
)
```

Listing B.18: PDDL description of an example problem in the *Custom Ball Factory* domain

```
(define (problem TestingScenarios)
  (:domain CustomBallFactory)

  (:objects
    ball1 - ball
    green blue - colour
    tl tr bl br - area
  )

  (:init
    (ball_free ball1)
  )
```

```
(:goal
  (and
    (striped ball1 tl green blue)
    (painted ball1 tr green)
    (painted ball1 bl blue)
    (striped ball1 br blue green)
    (inflated ball1)
    (ball_free ball1)
  )
))
```