THE UNIVERSITY *of* EDINBURGH

Multiagent Classical Planning

Matthew David Crosby

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2014

Abstract

Classical planning problems consist of an environment in a predefined state; a set of deterministic actions that, under certain conditions, change the state of the environment; and a set of goal conditions. A solution to a classical planning problem is a sequence of actions that leads from the initial state to a state satisfying the problem's goal conditions. There are many methods for finding solutions to classical planning problems, and a popular technique is to exploit structures that commonly occur. One such structure, apparent in many planning domains, is a breakdown of the problem into multiple agents. However, methods for finding and exploiting multiagent structures are not prevalent in the literature and are currently not competitive.

This thesis sets out to rectify this problem. Its first main contribution, is to introduce a domain independent algorithm for extracting multiagent structure from classical planning problems. The algorithm relies on identifying a generalisable property of agents in planning; namely, that agents are entities with an *internal state*, a part of the planning problem that, under a certain distribution of actions, only they can modify. Once this is appropriately formalised, the decomposition algorithm is introduced and is shown to produce identifiably multiagent decompositions over all of the classical planning domains used in the International Planning Competitions, even finding more detailed decompositions than are used by humans in certain cases.

Solving multiagent planning problems can be challenging because a solution may require complex inter-agent coordination. The second main contribution of the thesis is a heuristic planning algorithm that effectively exploits the structure of decomposed domains. The algorithm transforms the coordination problem into a process of subgoal generation that can be solved efficiently under a well-known relaxation in planning. The generated subgoals guide the search so that it is always performed by one single-agent subproblem at a time. The algorithm is evaluated and shown to greatly outperform current state-of-the-art planners over decomposable domains.

The thesis also includes discussion of the possible extensions of this work, to include the multiagent concepts of self-interested agents and concurrent actions. Results from the multiagent planning literature are improved upon and a new solution concept is presented that accounts for the 'farsightedness' inherent in planning. A method is then presented that can find stable solutions for a certain class of multiagent planning problems. A new method is introduced for modelling concurrent actions that allows them to be written without requiring knowledge of each other agent in the domain, and it is shown how such problems can be solved by a translation to single-agent planning.

Acknowledgements

Thank you to all who have helped along the way. My supervisors Michael Rovatsos and Ron Petrick for their insights and input. My examiners Gerhard Wickler, Anders Jonsson and Derek Long, whose comments have improved this document a lot. The agents factory and associates for providing discussions, proofreading and a work environment just productive enough to get this thing finished. To my family for being supportive of whatever I choose to do, even avenues representing a large sacrifice in current income in preparation for a large sacrifice in future income. Finally, to Martha for making everything alright, even through the failed ideas, the conference deadlines, and, most of all, the write-up period.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Matthew David Crosby*)

Table of Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Motivating Examples 4

1.2.1 The Robots Domain: Planning with Multiple Agents 5

1.2.2 The Parcels Domain: A Strategic Planning Problem 6

1.2.3 The Doors Domain: Planning with Concurrent Actions 7

1.3 Hypotheses and Contributions 8

1.3.1 Agent Decomposition 9

1.3.2 Strategic Multiagent Planning 11

1.3.3 Concurrent Action Constraints 11

1.3.4 Summary of Contributions 12

1.4 Thesis Structure 13

2 Background **15**

2.1 Classical Planning 16

2.1.1 STRIPS-style Classical Planning 17

2.1.2 The Planning Domain Definition Language 18

2.1.3 Multi-valued Planning Tasks 20

2.1.4 Heuristic Planning 22

2.2 Multiagent Planning 28

2.3 Shared-Goal Multiagent Planning 29

2.3.1 MA-STRIPS 30

2.3.2 Multiagent Decompositions 31

2.3.3 Multiagent Plan Synthesis 33

2.3.4 Concurrent Actions 35

2.4 Game-Theoretic Concepts for Planning 37

2.5 Strategic Multiagent Planning 38

2.5.1 Coalition-Planning Games 38
2.5.2 Equilibrium Concepts for Strategic Multiagent Planning . . . 39
2.5.3 Defining a solution concept for multiagent planning 42
2.5.4 Synthesising plans with self-interested agents. 45
2.6 Summary 46

3 Multiagent Multi-Valued Planning Tasks 47
3.1 Agents in MPTs 47
3.1.1 Example: Toy Robot Domain - Agents 50
3.2 Variables 52
3.2.1 Example: Toy Robot Domain - Variables 53
3.3 Actions 55
3.3.1 Example: Toy Robot Domain - Agent Variable Decomposition 62
3.3.2 Influenced and Influencing Actions 63
3.3.3 Example: Toy Robot Domain - Actions 64
3.4 Summary 66

4 Algorithms for Multiagent Classical Planning 67
4.1 Agent Decomposition 67
4.1.1 Causal Graphs 68
4.1.2 Causal Graphs and Agents 71
4.1.3 Agent Decomposition Algorithm 74
4.1.4 Decomposition Algorithm for the Robots Example 77
4.1.5 Summary 78
4.2 ADP - A Multiagent Heuristic Planning Algorithm 79
4.2.1 ADP - Algorithm 80
4.2.2 Heuristic Calculation 82
4.2.3 The Individual Heuristic Calculation 82
4.2.4 The Global Heuristic Calculation 83
4.2.5 Solving the Robots Domain 93
4.3 Summary 96

5 Evaluation 97
5.1 Experiment Design 97
5.2 Automated Agent Decomposition Results 99
5.2.1 Decomposition Structure 99

5.2.2 Decomposition Descriptions . . . 102
5.2.3 Decompositions by Domain . . . 105
5.2.4 Summary of Agent Decomposition Results . . . 115
5.3 Evaluation of ADP . . . 116
5.3.1 Analysis by Domain . . . 117
5.3.2 Collated Results . . . 129
5.3.3 Discussion: Design Decisions . . . 129
5.4 Summary . . . 132

6 Possible Extensions **133**
6.1 Multiagent Classical Planning with Self-Interested Agents . . . 134
6.1.1 Strategic Multiagent Planning . . . 134
6.1.2 Assumptions and Equilibrium Concepts . . . 135
6.1.3 A New Solution Concept . . . 139
6.1.4 Safe-MMPTs . . . 140
6.1.5 Solving Safe-MMPTs . . . 142
6.2 Concurrent Actions . . . 143
6.2.1 Types of Multiagent Interaction in Planning Environments . . . 144
6.2.2 Concurrent MMPTs . . . 145
6.2.3 Multiagent Planning Domain Set . . . 147
6.2.4 Solving Problems with Concurrent Action Constraints . . . 149
6.3 Summary . . . 154

7 Conclusion **157**
7.1 Future Work . . . 157
7.1.1 Decomposition Algorithm . . . 157
7.1.2 ADP . . . 158
7.1.3 Moving Away From Multiagent Classical Planning . . . 159
7.2 Contributions Revisited . . . 160
7.3 Summary . . . 161

A Further ADP Results **163**

Bibliography **171**

List of Figures

1.1 Multiagent Robots Problem 5
1.2 Strategic Planning Problem Requiring Cooperation 6
1.3 Strategic Planning Problem With Undefined Cooperation 7
1.4 Planning Problem With Concurrent Actions 8

2.1 Variables for the Parcel Domain 22
2.2 Relaxed Planning Graph of the Robots Domain 24
2.3 Causal Graph of the Robots Domain 27
2.4 Parcel Problem Requiring Cooperation 43
2.5 Parcel Problem with Unintuitive Stable Plan 43
2.6 Bridge Problem 44

3.1 Robot Grid World Domain 50
3.2 PDDL Actions for the Parcel Domain 50
3.3 Agent Decomposition Comparison 53
3.4 Variables for the Robot Domain 54

4.1 Agent Decomposition Algorithm Overview 68
4.2 Causal Graph for the Robot Problem 70
4.3 Modified Causal Graph for the Robot Domain 71
4.4 Relaxed Planning Graphs of Multiple Agents 86
4.5 A Second Layer of Planning Graphs 89
4.6 Proposition Objects in ADP Implementation 90
4.7 Extracting from a Relaxed Planning Graph 93
4.8 Robot Domain for Explaining Search Algorithm - Figure 1 94
4.9 Robot Domain for Explaining Search Algorithm - Figure 2 94
4.10 Robot Domain for Explaining Search Algorithm - Figure 3 95
4.11 Robot Domain for Explaining Search Algorithm - Figure 4 95

4.12 Robot Domain for Explaining Search Algorithm - Figure 5 96

5.1 Graph of Rovers Results 118
5.2 Graph of Satellites Results 121
5.3 Output of Relaxed Plan Generation for Logistics 1 124

6.1 Boat and Door Domains 147
6.2 Bridge and Switch Domains 148
6.3 Action Definitions with Concurrency Constraints 150
6.4 Maze Domain 151

List of Tables

3.1 Full Table of Action Types 59
3.2 Action Types in a MMPT 64

5.1 Table of Automated Agent Decomposition Results 100
5.2 Decomposition Descriptions 103
5.3 Rovers Domain Decomposition Results 106
5.4 Satellites Domain Decomposition Results 109
5.5 Logistics Domain Decomposition Results 110
5.6 Airport Domain Decomposition Results 112
5.7 Depot Domain Decomposition Results 113
5.8 Woodworking Domain Decomposition Results 114
5.9 Rovers Results 117
5.10 Satellites Results 120
5.11 Logistics Results 123
5.12 Airport Results 126
5.13 Depot Results 127
5.14 Woodworking Results 128
5.15 Combined ADP results 130

A.1 Driverlog Results 164
A.2 Floortile Results 164
A.3 Tpp Results 165
A.4 Transport Results 166
A.5 Zenotravel Results 167
A.6 Elevators Results 167
A.7 Mystery Results 168
A.8 Pathways Results 169
A.9 Storage Results 170

Chapter 1

Introduction

The main research area of this thesis is the extension of classical planning to include multiple agents. Classical planning involves the computation of a sequence of actions that can be used to manipulate an environment into a desired state. As such, it is an important component of Artificial Intelligence, modelling the human ability to plan ahead. The extension to multiple agents assumes that the actions in the domain are not just one large set, but distributed amongst the different agents in the domain.

In the classical planning literature there are domains that model multiple rovers navigating the surface of Mars, domains that model multiple vehicles transporting packages, and domains with multiple satellites coordinating to transmit data back to Earth. These are commonly used benchmark testing domains that are prevalent throughout the literature. Furthermore, they are clearly multiagent. However, the classical planning framework does not explicitly deal with the multiagent nature of these domains.

There are many sophisticated planning methods that focus on the structure inherent in certain planning problems. However, the structures used tend to be chosen because they are easily exploitable and commonly occurring as opposed to because they capture a problems inherent multiagent nature. The main idea behind this thesis is that, because the multiagent structure of certain domains is so self-evident and seemingly useful to a human observer, it should be possible to create algorithms to extract it, and to exploit it for faster planning.

This leads to the main contributions of the thesis: an automated method for extracting the multiagent nature inherent in certain classical planning problems and a heuristic planning algorithm for exploiting multiagent structures. The combination of these is shown to significantly improve planning times compared to state-of-the-art planners on

decomposable domains.

As the thesis is concerned with multiagent planning, it also covers the multiagent concepts most suited to being modelled in planning environments: self-interested agents and concurrent actions. It is shown how self-interested agents affect the planning process and a solution concept is developed that defines stable plans. An algorithm for finding stable solutions is then introduced that works on a non-trivial subset of planning problems. For concurrent planning, the problem of efficiently encoding concurrent action constraints is discussed and a new method introduced that scales better than existing methods as the number of agents increases. It is also shown how such problems can be solved with a translation to single-agent planning.

1.1 Motivation

This thesis sits at the intersection between automated planning (Nau et al., 2004; Russell and Norvig, 2003) and multiagent systems research (Wooldridge, 2001; Weiss, 1999). Through the consideration of strategic elements it also touches on parts of game theory (Gibbons, 1992; Osborne and Rubinstein, 1994).

Of the research areas covered, the main subject matter is automated planning. Automated planning is a broad research area with many different variations, extensions and applications. It fulfils an important role in general Artificial Intelligence research, providing a means for agents to reason deliberatively. However, the research area remains compact because classical planning, a simplified planning setting that defines the basic planning problem, acts as a focal point from which other approaches are extensions.

A classical planning domain (see Section 2.1) consists of a description of a world that starts in some predetermined state, a set of actions that can be performed to change the state of the world, and a goal state (or states). A solution to a planning problem is an ordered sequence of actions that will lead from the initial state to a goal state. This represents a fundamental problem, the solution to which can form the basis of any deliberative AI system. While classical planning can be seen as an essential research area, there are many possible extensions that are studied. This thesis is concerned with one such extension; namely, that to multiagent systems.

Multiagent systems research focusses on problems in which there are multiple interacting intelligent entities (called agents), each with autonomy over some part of the system. The multiagent systems research area is very diverse, covering any system that

has (or can have) multiple agents. This can include anything from solution concepts in abstract 2-person game theory problems to modelling thousands of agents in real-world emergency rescue scenarios. Game Theory is closely linked to multiagent research (Shoham and Leyton-Brown, 2008) and deals with the case where the agents are rational and self-interested. It is employed in this thesis as an extension of the basic multiagent planning problem to deal with coalitions of self-interested agents.

Multiagent approaches are becoming increasingly relevant as the size of computational systems increases. The advent of web-based approaches, multi-core processing and development in robot capabilities all require the incorporation of techniques from multiagent systems research. While these multiagent techniques cover a very broad area, this thesis takes the most relevant multiagent concepts and explores their application and impact on classical planning problems.

It has been argued that both multiagent systems and automated planning are important research areas amongst the wider AI community. However, this does not imply anything about the area between the two. The rest of this section argues that multiagent planning itself and, in particular, multiagent *classical* planning is an important area for research.

The extension of single-agent planning to multiagent planning is so natural that its complexities are often overlooked. Not only does it bring with it all the complexities associated with multiagent systems in general, but there are also unique problems with taking a multiagent approach to planning itself. The way in which different agents' plans interact, how actions are interleaved, and the capabilities and motivations of agents will vary between each research strand.

The number of possible multiagent assumptions and applications leads to a very diverse field with many seemingly incongruent approaches. One researcher may look at distributed planning with partial observability, durative actions and global cooperation while another may deal with centralised planning with full observability, probabilistic actions and self-interested agents with their own goals and the ability to form coalitions. It is currently an open question in the multiagent planning community of how to organise the field to allow for a stronger connection between the different approaches.

At the other end of the spectrum, Classical Planning includes a set of assumptions, each designed to keep the problem as simple as possible. It is a canonical version of the automated planning problem that can be used as a reference point for the other approaches in the field. The (single-agent) automated planning research area is, arguably, more diverse than its multiagent counterpart. However, it does not feel as separated

because most automated planning work can be linked directly back to classical planning in terms of the assumptions that it drops or the additions that it makes. That is why the area where classical planning meets multiagent systems is important for multiagent planning research. Classical planning, the element that ties the automated planning community together, is missing in the multiagent planning community.

Another argument for the importance of the research area is that there are many promising results in the wider multiagent planning literature that are directly applicable to multiagent *classical* planning. Recent work in multiagent planning has shown how the multiagent structure inherent to certain planning problems can be exploited to improve planning times (Brafman and Domshlak, 2008; Nissim et al., 2010). Solution concepts and planning methods have been proposed for solving planning problems in which agents are self-interested (Brafman et al., 2009; Jonsson and Rovatsos, 2011) and the complex problem of how to represent and plan with concurrent actions (Boutilier and Brafman, 2001) has been discussed. These results give rise to the topics addressed in this thesis.

A final argument for studying multiagent classical planning is the abundance of classical planning domains that are inherently multiagent. There are many domains that model a number of entities that need to coordinate their actions and work together to achieve certain goals. A human solving these problems would naturally break them down into separate problems for each agent and then work out a way to coordinate between them. As this solution method is so intuitive and useful from a human perspective it seems natural to attempt to replicate it with an automated process.

1.2 Motivating Examples

This section introduces example multiagent planning domains that further motivate the topics addressed in this thesis. The examples will be referred to throughout the thesis in order to explain and motivate the concepts and algorithms. The first shows why a multiagent approach to solving planning problems can be both natural and beneficial, the second shows how strategic concerns naturally arise in planning and the third introduces concurrent actions.

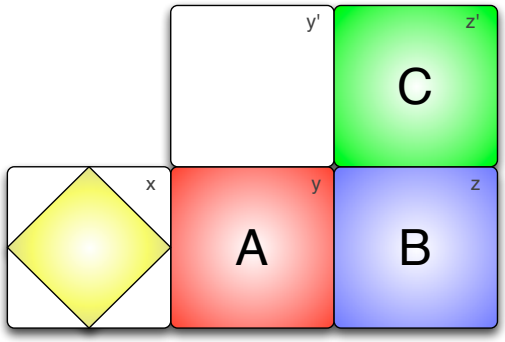

Figure 1.1: An example problem. The upper case letters represent robots that need to report to the square containing the diamond. The lower case letters are used to label the possible locations.

1.2.1 The Robots Domain: Planning with Multiple Agents

Figure 1.1 depicts a planning problem in which robots must navigate a gridworld in order to report to a particular location. Only one robot can occupy any given grid space at a time so they must coordinate their movements. In this example problem there are three robots A, B and C that must each reach the diamond location to report.

This domain has a very obvious agent decomposition, namely that each robot is a separate entity. A human solving this problem would almost certainly consider each robot's possible plans separately, and then work out how to coordinate them. It is natural to find a plan for A to get to the goal, move A out of the way, find a plan for B to get to the goal, move it out of the way and finally get C to the goal.

In contrast, single-agent planning approaches would take all possible actions in the domain as one large set and then try to order them to form a valid plan. Furthermore, standard heuristic techniques used in single-agent planning may consider moving any agent closer to the goal as a beneficial action thereby creating congestion around the goal location. More sophisticated planning methods may identify useful features of the problem, but not necessarily those that correspond to its multiagent nature.

It should be noted that even in this simple example, the final part of the plan involves coordination between all the robots. All three robots need to carefully coordinate their movements before C can finally make its way to the gaol. A key feature of the multiagent planning algorithm presented in this thesis is that it uses a heuristic approach to calculate required subgoals (such as a robot moving out of the way), without ever explicitly dealing with the full coordination problem. The subgoals are then used to

guide the search. While not useful in the worst case, this method is empirically shown to be effective in the majority of existing multiagent domains used in the evaluation.

For this domain, finding a multiagent decomposition, even though it makes the complex coordination problem between the agents explicit, could clearly be beneficial, However, it is not immediately obvious which components of the domain give rise to its multiagent structure and it is certainly not obvious how such components could be generalised across all planning problems.

This leads to the two main research questions that are answered in the thesis. The first asks: is it possible to create an automated process to extract, where it exists, the multiagent structure of planning domains? The second question asks: how can the multiagent structure of a domain be exploited for faster planning? Can the human process for solving the example problem be mirrored (in some sense) in a multiagent planning algorithm and how successful is such an algorithm in the general case?

1.2.2 The Parcels Domain: A Strategic Planning Problem

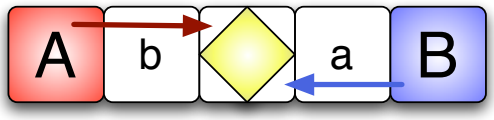

Figure 1.2: Agent A (red) has to deliver parcel a to the depot (yellow diamond) while agent B (blue) has to deliver parcel b to the same depot.

Figure 1.2 depicts a planning problem in which two delivery agents must deliver a parcel to a central depot location. An uppercase letter represents an agent, a lowercase letter represents a parcel associated with that agent and the yellow diamond represents a depot where each agent has to deliver its parcel(s). Each agent can either move, pickup or drop a parcel, and multiple agents can occupy the same space in the grid.

Logistics domains are commonly used in planning research and have a very natural multiagent interpretation. If delivery agents are associated with individual parcels then it makes sense to assume that they prefer the delivery of that parcel over other parcels that may be in the domain. This is visible in real-world logistics domains where multiple delivery companies are concerned with fulfilling their customer's orders.

In order to model this real-world scenario, it makes sense that the overall goal of the planning problem is distributed amongst the agents. Robot A has to deliver parcel a,

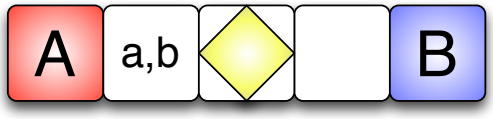

Figure 1.3: Agent A (red) has to deliver parcel a to the depot (yellow diamond). Agent B (blue) has to deliver parcel b to the same depot.

while robot B has to deliver parcel b. This means that the agents will have preferences over the possible plans in the domain. After all, they will prefer a plan that completes their goal to one that does not, perhaps regardless of the number of other agents' goals that the plan completes.

Agent A's individual solution to the problem shown in Figure 1.2 would be to move to parcel a, pick it up and then move to depot a and drop it. This would take six actions (four moving and two picking up/dropping). Similarly, it would take six actions for agent B to deliver its parcel. However, it should be obvious that if the agents cooperated and each delivered the other's parcel, then they would both benefit. In this case the globally optimal plan in the lowest cost plan for both agents, but this is not always the case.

Figure 1.3 depicts a problem where the globally optimal plan is not the lowest cost plan for both agents. In this problem, it is globally optimal for A to deliver both the parcels; however, A has no reason to spend the extra actions to deliver B's parcel unless B will pay him to do so. The 'correct' solution to this problem depends on the assumptions that are made about the agents.

This example domain shows how strategic considerations naturally arise when dealing with multiple agents. It also hints at the complexities involved when planning for them. There are many different assumptions that can be made about how the agents can interact or coordinate and the preferences they can have over the possible global plans. This is an important area of multiagent planning; but, due to the many possible assumptions and complexities, one for which there are few conclusive results.

1.2.3 The Doors Domain: Planning with Concurrent Actions

Figure 1.4 (page 8) shows a problem involving concurrent actions. As with the previous domains, the robots A and B must navigate to the goal location. They can move to an adjacent grid square and jointly occupy locations. However, in this domain the agents perform their actions concurrently. The door is only large enough for one agent to pass

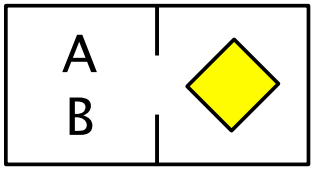

Figure 1.4: Agents A and B have to navigate to the goal area. However, they must act concurrently and only one can pass through the door at a time.

through at a time so if both agents attempt to move through it, then neither will succeed.

The solution to the problem is for one agent to wait while the other passes through and then for that agent to pass through on their own. This problem, along with its solution, requires a formalism that can represent the concurrent plans and their constraints. The constraint that the door can only be used by one agent at a time needs to be included along with a plan representation that sets each agent's actions side by side.

This example shows a concurrent action constraint with a negative consequence; it prohibited certain actions from being performed simultaneously. It is also possible to model concurrent action constraints that define positive interactions. For example, if two agents must lift a table together, then their respective actions must be performed simultaneously. As with strategic considerations, the addition of concurrent actions drastically increases the complexity of the multiagent planning problem. This thesis discusses concurrent actions in the area as close to classical planning as possible and looks at how they can be efficiently encoded into a classical planning problem definition.

1.3 Hypotheses and Contributions

There are four main topics for which this thesis makes contributions. The first two are the most closely related to classical planning and represent the main contributions of the thesis. They involve both finding, and planning for, agent decompositions. The final two topics are included as a discussion about extensions of the core multiagent planning techniques presented in this thesis; the first of these concerns self-interested agents, while the latter involves concurrent actions.

1.3.1 Agent Decomposition

It was already mentioned that recent results have shown how the multiagent structure in certain classical planning domains can be exploited for more efficient planning. To do this, problems are divided into interacting subproblems, such that each agent can use only a subset of the available actions. However, these methods generally rely on an *a priori* multiagent planning problem specification created by a human expert. This can be a time consuming process and requires familiarity with a domain. An automated process for finding agent decompositions would allow for multiagent planning algorithms to be applied to existing domains without requiring human input. It would also mean that future multiagent planning problems could be written using the existing and widely-used single-agent method.

A divide and conquer approach has been an influential concept in automated planning since its advent (Sacerdoti, 1973). Because of this, there are many methods that exist for decomposing planning domains (eg. Durfee and Lesser (1991)). However, the decomposition of planning domains specifically into multiple agents, that correspond with the concept of agents taken from multiagent systems research, is less well explored. While some methods exist (Nissim et al., 2012), these are heuristic methods based on minimising interactions as opposed to a focus on multiagent structure.

It has been argued that there are many existing domains which have a pronounced multiagent structure. Because these multiagent structures appear so naturally to a human observer, it should be possible to identify and exploit the specific elements of these domains that cause them to have a multiagent interpretation. This thesis puts forward the idea that, under certain decompositions of actions, agents can be identified due to the fact that they have an *internal* state, a part of the environment that only they have control over.

This leads to the first hypothesis of the thesis:

> **Hypothesis 1:** It is possible to design a domain independent algorithm that can find the 'multiagent structure' of classical planning problems which can then be exploited for faster planning with a multiagent search algorithm.

The decomposition algorithm isolates parts of the domain that can represent internal states of agents and these are used to define the agents themselves and create a decomposition. The algorithm is tested over all the classical planning domains used in the International Planning Competitions and is shown to find multiagent decompositions in all cases that appear to be multiagent in nature.

The algorithm can be used as a method for creating agent decompositions from planning problems written in the standard (most common) single-agent style. This allows the large body of problems used in single-agent planning to be seamlessly integrated into multiagent planning work. It also allows for multiagent planning problems to be written in the well-known single-agent planning problem domain definition language. Furthermore, the results provide a deeper insight into the underlying structure of existing benchmark planning problems.

While the previously mentioned results in multiagent planning show improved performance by exploiting multiagent structure, this is not compared with performant single-agent planners. Naturally, if multiagent decompositions of planning domains are to be useful, it should be possible to exploit them for faster planning.

This gives rise to the second hypothesis of the thesis:

> **Hypothesis 2:** It is possible to design a multiagent planning algorithm that successfully exploits the multiagent structure of decomposed planning problems to plan faster than state-of-the-art single-agent planners.

This thesis presents a multiagent planning algorithm and implementation (ADP) that improves on planning performance over current state-of-the-art single-agent planners. ADP is a heuristic planning algorithm that utilises the famous 'no delete lists' heuristic from single-agent planning work (Hoffmann and Nebel, 2001). ADPs improved performance is, of course, restricted to domains for which a multiagent decomposition can be found, but this set is larger than the number of domains considered in the standard multiagent planning literature. The algorithm is tested over all domains from the International Planning Competition. It does not find a plan faster in every single case, but is shown to outperform the competing single-agent planners on the vast majority of problems for which a decomposition can be found.

The decomposition algorithm and ADP can be combined to form a planner that can solve the full set of problems faster than existing planners. If the decomposition algorithm returns a decomposition then ADP is used, otherwise the current best single-agent planner is invoked. The decomposition algorithm has negligible run time compared to plan search time so this combination planner represents an improvement over the state-of-the-art in solving classical planning problems. Around one third of the tested domains had an agent decomposition, so the improvement is significant. The full results are discussed in Chapter 5.

The contributions mentioned so far can be thought of as multiagent decomposition methods for centralised classical planning and, while they utilise ideas from multiagent

systems research, do not necessarily include components that make the system multiagent. This thesis therefore also covers two extensions of this basic multiagent classical planning setting: planning with self-interested agents and planning with concurrent actions.

1.3.2 Strategic Multiagent Planning

Strategic multiagent planning is an interesting research area because there is a contradiction between the standard assumptions made in game theory and the nature of planning. First of all, an interesting multiagent planning domain contains many possible interactions between agents, with agents' actions affecting what is possible for other agents in the domain to achieve. To solve most problems requires some form of cooperation between agents, which naturally suggests modelling coalitions, sets of agents that agree to work together to improve their individual utility. Secondly, the standard approach to coalitions in game theory ignores the long-term consequences of deviations. This goes against the fundamental property of planning agents, the ability to look ahead, which is called 'farsightedness'.

The natural connection between strategic multiagent planning and coalitions has led to the study and definition of Coalition planning games (Brafman et al., 2009). However, the solution concepts employed in this area of research are problematic (see Section 2.5.3). This thesis discusses the problematic nature of coalitions in planning and introduces a new solution concept that is shown to improve on the results in the literature. However, it was not possible to find an algorithm that can find stable solutions in the general case. Instead, this thesis defines a subclass of planning problems for which a solution can be found and introduces an algorithm that can be used to find such solutions. This subclass of problems is large enough to include the example presented in Section 1.2.2), and any similar problem of increased size.

1.3.3 Concurrent Action Constraints

Probably the most important design decision in multiagent planning problems is the choice of how actions can interact. In the preceding, it has been assumed that actions between agents are interleaved and that there are no conflicts caused by the ordering of such actions. This is the approach most commonly taken in the multiagent planning literature with many papers not explicitly stating the assumptions they make about action interactions.

However, multiagent planning brings with it the ability to model concurrent actions such as lifting a table together or simultaneously passing through a small doorway. This is addressed in (Boutilier and Brafman, 2001) which models concurrent action constraints by modifying action definitions. This has the downside of requiring knowledge of every possible action in a domain before a new constraint can be added. If a new agent is to be added, then its actions can only be defined in terms of all the actions held by each other agent in the domain. This thesis introduces a new method for defining concurrent action constraints by relating them to objects (or more accurately, their affordances) as opposed to actions. This allows for a much simpler and more efficient definition process in large domains. Unfortunately, it is beyond the scope of current research to provide efficient methods for planning in all but the simplest domains with concurrent actions. Instead, it is shown how to translate concurrent action domains in to problems solvable with existing planning technology so that, in the future, this can be used as a comparison for multiagent techniques.

1.3.4 Summary of Contributions

This section collates the contributions of the thesis, including those discussed so far and also the smaller contributions that are covered on the road to answering the main research questions of the thesis.

- **Formalisation of MMPTs:** Multiagent Multi-Valued Planning Tasks (MMPTs) are a formalisation for multiagent planning that remains close to the assumptions of classical planning. Agents are defined by the set of variables in the domain that represent their internal state and there is a set of variables that defines the environment the agents are acting in. Defining agents in this way coincides with the multiagent systems community definition of agents as autonomous entities acting and interacting in some environment. This is a new way of looking at agents in planning domains and this shift in focus allows for the creation of agent decomposition algorithm that finds decompositions that are easily recognisable as multiagent.

- **Agent Decomposition Algorithm:** An algorithm is introduced that can find agent decompositions (valid MMPTs) given single-agent classical planning problems as input. This algorithm is shown to conform to the expectations of the breakdown of agents in all domains that it is tested on. These multiagent decompositions can then be used as input for a multiagent planning algorithm.

- **ADP, an Agent Decomposition Planner** An algorithm (ADP) is presented for planning with MMPTs. ADP breaks down the problem so that search is always performed using single-agent subproblems. In between single-agent searches, the coordination problem is solved in a relaxed version of the search space that can efficiently distribute goals to agents even in domains that require large amounts of interaction. The algorithm is shown to greatly outperform existing planners on decomposable domains.

- **Extensions** For strategic multiagent planning, a new solution concept for coalition planning games is introduced that takes into account a planning agent's ability to look ahead and is shown to have more intuitive results than the solution concept taken from the literature. Also, a subclass of strategic planning problems is defined that can be solved using modifications of existing planning heuristics. This subclass is broad enough to include the logistics example presented in the previous section.

  For planning with concurrent actions, a method for defining concurrent action constraints is presented that allows for problem domains to be specified without knowledge of the actions belonging to other agents. Also, a set of planning domains is introduced that covers the different ways in which agents can interact in a multiagent environment. These domains can be combined to create any combination of the different interaction types required.

The results regarding agent decompositions and ADP were presented, in an abridged and earlier form, in the paper Automated Agent Decomposition for Classical Planning that was published in ICAPS 2013 (Crosby et al., 2013). An early version of the results regarding coalition planning games was published in the short paper Heuristic Multi-agent Planning with Self-Interested Agents at AAMAS 2011 (Crosby and Rovatsos, 2011).

1.4 Thesis Structure

The rest of the thesis is structured as follows: The next chapter (Chapter 2) surveys the related literature and provides the relevant background information for the work presented in this thesis. It is split into three main parts: single-agent classical planning, completely cooperative multiagent planning and strategic multiagent planning. Chapter

3 presents the MMPT formalism used as a basis for the planning methods presented in the rest of the thesis.

The remaining chapters present the algorithms and evaluation for the thesis. Chapter 4 introduces an algorithm for automated agent decomposition and presents ADP, an algorithm for solving such decomposed problems. Chapter 5 presents the evaluation of the preceding algorithms while Chapter 6 looks at the extension of MMPTs to include strategic concepts and concurrent action constraints. Finally, Chapter 7 provides the conclusion and discusses possibilities for future work.

Chapter 2

Background

This chapter surveys the related literature and introduces the relevant background information for the rest of the thesis. Due to the nature of the thesis topic, this includes material from Automated Planning, Multiagent Systems and Game Theory. While there is a large amount of loosely related literature, this chapter focuses on the works directly relevant to the results presented in the thesis.

It could be argued that the entire multiagent planning literature is relevant to this thesis and therefore should be discussed. However, this area is far too large to cover in its entirety. As was discussed in the introduction, this thesis deals with multiagent planning at an abstract level, with a focus on using the multiagent nature inherent in certain problems as a tool for more efficient plan synthesis. This is one step removed from more real-world multiagent planning considerations such as plan execution, coordination and merging such as can be found in Ephrati and Rosenschein (1993a).

The most relevant literature from single-agent planning that is discussed in this chapter includes the Planning Domain Definition Language (McDermott, 2000), the MPT planning representation (Bäckström and Nebel, 1995) and heuristic methods based on the 'no delete lists' relaxation (Bonet and Geffner, 1999; Hoffmann and Nebel, 2001). From the multiagent planning literature, the most important areas for this thesis are multiagent plan decomposition (Nissim et al., 2012), cooperative and strategic multiagent plan synthesis (Brafman and Domshlak, 2008; Brafman et al., 2009), and planning with joint actions (Boutilier and Brafman, 2001).

The chapter is structured as follows. Firstly, Section 2.1 introduces classical planning and the relevant work from the automated planning community. After this, Section 2.2 looks at the extension of classical planning to multiagent planning. This leads to the multiagent planning part of this chapter, which is split into two sections. The

first, Section 2.3, discusses multiagent planning approaches for which there is a shared goal and complete cooperation between the agents. The second, Section 2.4, looks at strategic multiagent planning approaches where each agent has its own goal set and its own preferences over possible plans. Finally, Section 2.6 concludes the chapter.

2.1 Classical Planning

A basic planning domain consists of a world that starts in some predetermined state, a set of actions that can be performed to change the state of the world, and a goal state (or set of states). Classical planning (Weld, 1999) is a core case of the general automated planning problem and is characterised by a number of simplifying assumptions. The list presented here is adapted from that presented in Nau et al. (2004).

- Finite: The system has a finite set of states.
- Observable: The system, including the initial state, is fully observable.
- Deterministic: All actions are deterministic.
- Static: The state of the world will not change without an action being performed.
- Simple Goals: Goals are either met or not met by a plan. There is no preference ordering over goals.
- Sequential Plans: Plans are linearly ordered, finite sequences of actions.
- Implicit Time: Actions have no duration and there is no explicit representation of time.
- Offline Planning: If a solution exists it can be found offline.

This thesis focuses on maintaining the classical planning assumptions as much as possible. All problems in this thesis will be finite, observable, deterministic, static, (mostly) sequential plans with implicit time and offline planning. In Chapter 6.1, the assumption of simple goals is dropped when discussing strategic multiagent planning, where agents may have separate goal sets and preferences over plans. Similarly, the assumption of sequential plans is relaxed when considering concurrent action constraints in Chapter 6.2.

2.1.1 STRIPS-style Classical Planning

The multiagent planning literature generally utilises a planning formalism with a STRIPS-style action representation (Brafman and Domshlak, 2008; Jonsson and Rovatsos, 2011). This is also the approach taken in this thesis. Therefore, this section introduces a formalism for classical planning with STRIPS-style actions (Fikes and Nilsson, 1971).

A classical planning problem is defined over a logical language $\mathcal{L}$. This language contains a finite set of predicates ($\mathcal{L}_p$), an infinite set of variables ($\mathcal{L}_v$) and a finite set of constants ($\mathcal{L}_c$). Constants are used to represent each of the possible objects in the domain. Variable symbols may represent an arbitrary constant from the domain and are unlimited in number (though in what follows only a finite amount can appear in any proposition). A variable may range over any number of constants in the domain. Predicate names are used to define relations between the objects in the domain.

The example domain given in Section 1.2.1 may contain, for example, the predicate `at(robot_a, location_x)`, a binary predicate that establishes a relationship between its two arguments `robot_a` and `location_x`. When instantiated with the constants `robot_a` and `location_x` it is interpreted to mean that `robot_a` is *at* `location_x`.

The set P is used to represent the set of all atomic formulae of language $\mathcal{L}$. An atomic formula in $\mathcal{L}$ is a single predicate of variables and/or constants. The set $P_{\mathcal{G}}$ contains only the ground atoms (containing only constants) in P. States of the planning domain are represented by a subset $S \subseteq P_{\mathcal{G}}$. A ground atom p holds in the state of the world represented by S as long as $p \in S$. The standard planning formalism employs the closed–world assumption, so that if $p \notin S$ then p does not hold in the state of the world represented by S.

An operator is a tuple of the form $\langle n, pre, eff \rangle$ where:

1. n represents the name of the operator,
2. $pre = \langle pre^+, pre^- \rangle$ is the set of positive and negative preconditions, and
3. $eff = \langle eff^+, eff^- \rangle$ is the set of positive and negative effects.

Each set pre^+, pre^-, eff^+ and eff^- is a subset of P. An operator may contain variables and have a large number of possible groundings (where the variables are instantiated with constants in their range). A ground operator is called an action.

An action a is applicable in a state S, if all positive preconditions of a belong to S and there are no negative preconditions of a in S. The effect of an action a is that each

positive effect is added to the state S and each negative effect is removed from S. If a negative effect is not in S then it is ignored and treated as already removed. If an action contains the same ground atom in its positive and negative effects then, by convention, that effect remains in the subsequent state. In other words, if a is applicable in state S then it will change state S into $(S \setminus eff^-) \cup eff^+$. Extensions of the classical planning formalism allow action preconditions and effects to contain quantifiers and conditional statements.

The goal of a planning problem contains ground atoms from $P_{\mathcal{G}}$. The goal G of a planning problem is reached in state S if $G \subseteq S$. This means that there will likely be multiple goal states of the planning problem. For example, in the problem shown in Figure 1.1, the goal is for each robot to have *reported* at the starred location. Each state in which they have reported, regardless of their current positions, is a goal state. In all states with $G \not\subseteq S$ the goal has not been reached.

Putting all this together it is possible to define a *classical* planning problem.

Definition A (single-agent) *classical* planning problem over a language $\mathcal{L}$ is a tuple $\Pi = \langle P, O, I, G \rangle$, where:

1. P is a finite set of atomic formulas of $\mathcal{L}$,
2. O is the set of operators,
3. $I \subseteq P_{\mathcal{G}}$ is the initial state of the problem, and
4. $G \subseteq P_{\mathcal{G}}$ is the goal of the planning problem.

A solution to a planning problem Π is a *plan* $\pi = [a_1, \ldots, a_n]$, an ordered sequence of actions that can be executed in sequence such that S is the state reached from applying the actions in π in order and $G \subseteq S$ holds, i.e. a goal state has been achieved.

2.1.2 The Planning Domain Definition Language

While the preceding defines the classical planning problem, a language is still needed in which planning domains can be efficiently written and used as input for planning algorithms. The planning domain definition language (PDDL) (McDermott, 2000) is the language used to define the domains that appear in the International Planning Competition, a biennial event designed to compare current state-of-the-art planners (see International Planning Competition (2008)). These domains are used as the main testbed for the evaluation section of this thesis.

Modern versions of PDDL have capabilities extending far beyond those of the classical planning problem presented in the last section. The version presented here is not the most recent, but contains enough structure to cover the domains used in the evaluation of this thesis. PDDL will be used to present the example problems and techniques used in this thesis.

An important feature of PDDL, that allows for the efficient writing of domains, is typing. Constants in PDDL definitions can be given a number of types which are used to limit the scope of variables. For example, the objects `robot_a`, `robot_b` and `robot_c` could all be given the type `robot` so that the variable `?r - robot` is fixed to only be instantiated as one of the three robots. This is simply a method for improving the writing and readability of domain and problem definitions and does not introduce any new capabilities. Any typed domain can be rewritten without types by adding unary predicates for each type along with appropriate preconditions for each action that uses that type.

A PDDL planning domain is split into two separate files. The first, the domain file, lists the types, predicates and operators of the domain. The second, the problem file, contains the objects, initial state and goal state for a specific problem. In general, there can be multiple different problem files for use with a single domain file.

The following lists the elements of the PDDL domain and problem files in the order in which they appear:

Domain File

- **Types:** Types in PDDL can have a subtype structure. The text `agent parcel - object` would define the types agent, parcel and object with agent and parcel being subtypes of the type `object`. All constants of type `agent` also are of type `object`.

- **Predicates:** The predicates (members of $\mathcal{L}_P$) of the domain are defined in the domain file. An example predicate definition is `(connected ?x - location ?y - location)` which would represent that two constants of type `location` are connected.

- **Operators:** An example operator (labeled as an action in PDDL) would be:

```
(:action move
:parameters (?a - agent ?x - loc ?y - loc)
```

```
:precondition (and (at ?a ?x) (free ?y) (connected ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x))
               (free ?x) (not (free ?y)))
)
```

The negated preconditions and effects represent the elements of pre^- and eff^- respectively.

Problem File

- **Objects:** The objects in the domain are given by a (potentially typed) list of constants. For example, `robot_a robot_b robot_c - robot` defines the objects `robot_a` through `c` and says that they all have type `robot`.
- **Initial State:** The initial state is written as a list of ground atoms that are true at the start of the problem.
- **Goal State:** The goal state is written as a list of ground atoms that are required to be made true to solve the planning problem.

PDDL is the format used as the input for the algorithms and is also used to present the actions and predicates included in the examples used throughout the thesis. It provides an efficient way to write down classical planning problems which must then be parsed by the planning algorithm. Once parsed, there are many techniques used in planning to create a more efficient representation of the problem that can be exploited during search.

2.1.3 Multi-valued Planning Tasks

Most planners parse the PDDL input files and immediately ground all operators to form a large set of actions. The most basic planning representation consists of this set of actions along with the initial state and goal state for the domain. However, modern planners perform further operations on the initial problem to create a more detailed planning representation with properties that can be exploited to plan more efficiently. Much of the work in this thesis utilises the Multi-valued Planning Tasks representation (Helmert, 2006).

Multi-valued Planning Tasks (MPTs) are based on the SAS+ planning representation (Bäckström and Nebel, 1995; Jonsson and Bäckström, 1998). The idea is to determine

sets of mutually exclusive ground atoms which are used to form variables. By mutually exclusive it is meant that only one of a variable's possible values can be true at any given time during the execution of a plan. The extra knowledge that each variable may only have one value at a time can then be exploited during planning to guide search.

Definition A *multi-valued planning task (MPT)* is a 5-tuple $\Pi = \langle V, I, G, A, O \rangle$ where

- V is a finite set of state variables v, each with an associated finite domain D_v,
- I is a state over V called the initial state,
- G is a partial variable assignment over V called the goal,
- A is a finite set of (MPT) axioms over V, and
- O is a finite set of (MPT) operators over V.

A *partial (full) variable assignment* is a function f from $V' \subseteq V$ ($V' = V$) such that $f(v) \in D_v$ for all $v \in V'$. A *state* is a full variable assignment. *Axioms* are triples of the form $\langle c, v, d \rangle$, where c is a partial variable assignment called the condition or body of the axiom, v is a variable, and $d \in D_v$ is called the derived value for v. An *operator* $\langle pre, eff \rangle$ consists of a partial variable assignment *pre* over V called its precondition, and a finite set of effects *eff*. Effects are triples $\langle cond, v, d \rangle$, where *cond* is a (possibly empty) partial variable assignment called the effect condition, v is a fluent called the affected variable, and $d \in D_v$ is called the new value for v.

Variable domains may contain the value $\perp$ to represent that none of the other atoms in the domain of the variable hold. For example, a variable may be $V = \{$`(free, loc1)`$, \perp\}$ which would represent whether location 1 is free or not.

MPT's are calculated by analysing actions and axioms to find atoms that form invariant sets (Edelkamp and Helmert, 1999). This process involves searching the set of positive effects for elements that can be paired with elements from the negative effects to ensure that only one is true at a time. The exact details of this calculation are not relevant to the results presented in this thesis except that the returned representation minimises the number of variables while maximising the size of the variable domains.

The variables used in MPTs prove to be very useful constructs for dealing with agents in multiagent planning approaches. A variable effectively categorises the state of a particular part of the domain. It will be argued later that an agent can be characterised by the part of the environment that it has autonomy over, therefore separating it from the

```
//A variable for each location stating if it is free
V1 = {⊥,(free loc1)},...,V5 = {⊥,(free loc5)}
//A variable for the location of each agent
V6 = {(at A loc1),(at A loc2),(at A loc3),(at A loc4)}
V7 = {(at B loc2),(at B loc3),(at B loc4),(at B loc5)}
//A variable for the location of each package
V8 = {(at a loc1),...,(at a loc5),(holds A a),(holds B a)}
V9 = {(at b loc1),...,(at b loc5),(holds A b),(holds B b)}
```

Figure 2.1: The set of state variables for the parcel domain introduced in Section 1.2.2. Each variable may only have one value at a time. A state is an assignment of a value to each variable.

other agents in the domain. This means that a variable in an MPT can form a building block from which the agents in a domain can be determined. This process is discussed in detail in Chapter 3.

This section has so far introduced classical planning, a language for writing planning domains, and MPTs, a planning representation that is utilised throughout the thesis. After writing a planning domain, parsing it and then creating a useful representation, the next step is try and find a plan. There are two main approaches towards plan search: optimal and heuristic. In optimal planning the aim is to find the lowest cost plan that achieves the goal, while in satisficing planning the aim is to return any plan that achieves the goal, regardless of its cost. Optimal planning is preferred in environments where the cost of the plan is important and which are sufficiently small, satisficing planning is preferred when the aim is to find a plan as quickly as possible. The methods presented in this thesis are for satisficing planning and based on heuristic planning techniques used in single-agent planning. These are presented in the next section.

2.1.4 Heuristic Planning

STRIPS planning is known to be PSPACE-complete (Bylander, 1994), so many planners focus on heuristic techniques. These techniques, while inefficient in the worst case, allow for efficient planning in the majority of commonly used planning problems. The success of these techniques perhaps comes from the fact that many human-implemented planning problems have inherent structure that can be exploited. This structure exists

due to the way in which a PDDL operator can encode a large number of instantiated ground actions (Section 2.1.2). These actions are linked in structure by the definition of the operator, meaning that complex domain with many ground actions, will naturally have a lot of similarities between the intended semantics of actions.

This section introduces two separate planning systems. The first, Fast-Forward, uses the 'no delete lists' relaxation which is the relaxation used by the algorithms presented in this thesis. The second, Fast-Downward, is a recent planning system whose implementation is used as a basis for the implementation of the algorithms presented in this thesis.

2.1.4.1 The Fast-Forward planning system and Relaxed Planning Graphs

The Fast-Forward (FF) planning system (Hoffmann and Nebel, 2001) is a very successful heuristic planner that has won numerous awards at the International Planning Competition. It uses a very simple heuristic that has a fast calculation and also returns a relatively accurate estimate.

The basic idea of the heuristic calculation is to relax the planning problem by ignoring all elements of eff^-; this is called the 'no delete lists' relaxation. This idea was preceded by earlier work such as Sacerdoti (1973) which considered, as a highest level of multiple possible abstractions, planning actions with no preconditions.

Under the 'no delete lists' relaxation, actions can only increase the size of a state. This means that applying an action always leads to a new state with at least as many applicable actions. Therefore, when planning under the relaxation, actions can be tried as soon as they become applicable without fear of any negative consequences. FF utilises this fact by calculating heuristic values using planning graphs. A depiction of the planning graph calculated from the initial state of the Robots domain from the introduction (Figure 1.1, page 5) is shown in Figure 2.2.

A planning graph is created by repeatedly applying all possible actions to a state until no more actions can be added or a goal state is reached. The first layer contains the propositions present in the initial state. At each time step, every applicable action is found and added to the graph if it has not been used before. All the positive effects of the applicable actions are then added to form a new state. At layer i the proposition layer consists of all propositions that can possibly be reached in i time steps and the action layer consists of all actions that are applicable given those propositions.

For example, Figure 2.2 shows three applicable actions in the initial state, robot A can move up or left and robot C can move left. Adding the positive effects of these

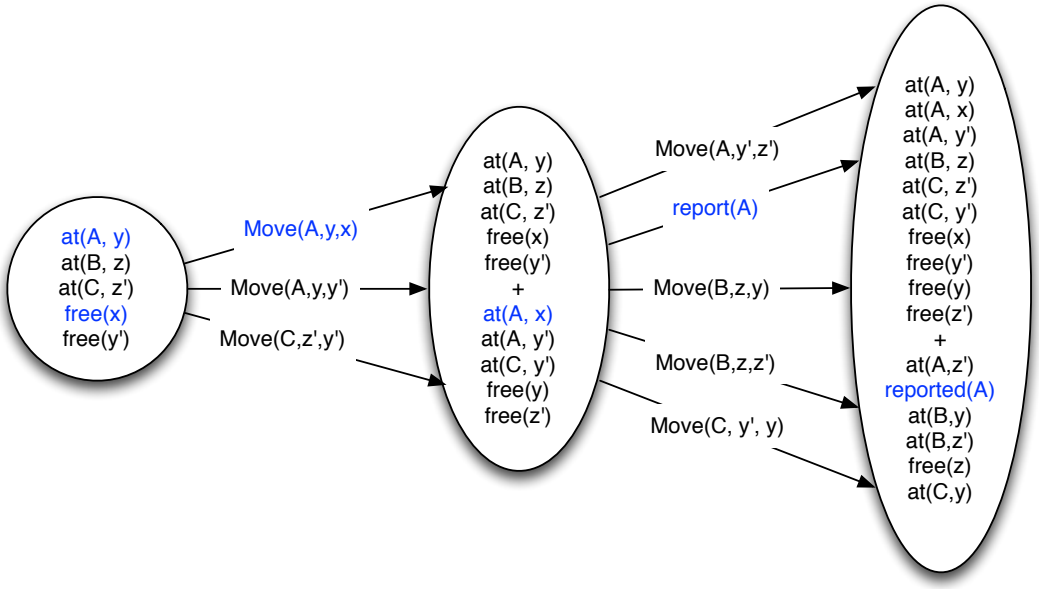

Figure 2.2: The first few layers of the relaxed planning graph for the Robots problem shown in Figure 1.1. The blue text shows how a relaxed plan can be extracted from a goal proposition back to the initial state.

actions results in a state that does not make sense in the full planning problem. In the resultant state robot A is in three different locations and location y is both free and contains robot A. It does not matter that this state is impossible in the unrelaxed problem, what is important is that the relaxed problem is easy to solve and its solution has some correlation with the solution to the full problem.

Any goal propositions reachable in the full problem are guaranteed to appear in the relaxed planning graph. This is because the relaxed problem is strictly easier than the full problem. This allows the relaxed planning graphs to be used to generate admissible heuristic values for each state. An algorithm that calculates heuristic values by adding together the depth at which goal propositions occur in the relaxed planning graph would already be a useful planner (Bonet and Geffner, 2001).

However, it is possible to find an even more accurate heuristic estimate. This is achieved by extracting a relaxed plan from the relaxed planning graph. For each goal proposition, a route can be traced backwards through the graph to the initial state. For each proposition to be extracted, an action that achieves it is added to the relaxed plan. For each action added to the relaxed plan, the propositions in its preconditions are added to the extraction list. Eventually this process leads back to the initial state. The collection of all actions that appear in this process is the relaxed plan to the goal.

In Figure 2.2, the text highlighted in blue shows this process for the goal proposition `reported(A)`. It would result in the relaxed plan

```
[move(A,y,x), report(A)].
```

This is coincidentally a valid plan in the full problem, but this is not always the case. For the full goal of the problem the relaxed plan would be

```
[move(A,y,x), report(A), move(B,z,y), move(B,y,x), report(B),
              move(C,z',y'), move(C,y',y), move(C,y,x), report(C)].
```

This plan is not executable in the full problem but gives a heuristic value of 9, the number of actions in the plan, which can help guiding search. The heuristic value will generally be smaller in states where the robots have moved closer to the goal square and in states where some robots have already reported.

One additional property of the heuristic calculation is that it can also provide a list of 'helpful actions' to further guide the search. If an action appears in the first layer of the relaxed planning graph, and contributes to at least one new fluent in the next layer that is used in the path to the goal, then it is added to a set of 'helpful actions' that are prioritised by the overall search algorithm. In the example, the action `move(A,y,x)` provides a useful fluent in the next layer, while the action `move(A,y,y')` does not. This means that the heuristic value for the state resulting from applying the action `move(A,y,x)` will be found before that for action `move(A,y,y')`. If the former state results in a lower heuristic estimate than the current best heuristic estimate, then the latter path may be pruned entirely from the search tree.

The FF search process is split into two parts: enforced hill climbing (EHC) and A* Search. The former search algorithm is a greedy search using the heuristic value of states calculated under the 'no delete lists' relaxation of the planning problem. The algorithm performs breadth-first search for a successor with a better (lower) heuristic value. If no improved successor can be found then the EHC part of the search algorithm returns failure. This can happen even when a plan exists for the problem.

If EHC search fails then A* search is invoked using the same heuristic calculation to guide search. As A* search will back-track indefinitely if it ends up in a dead end, a plan will always be returned if one exists. Empirically, it has been shown that EHC search is good enough for a large number of existing planning domains.

As will be shown in Chapter 4.2, the 'no delete lists' heuristic' translates smoothly to a multiagent approach. The same algorithms for creating planning graphs and extracting

plans can be used. Furthermore, the final state of a completed planning graph forms a concise representation of the information required to be passed between agents to solve the multiagent coordination problem.

2.1.4.2 Fast-Downward and Causal Graphs

The multiagent planning algorithms presented in this thesis are implemented as extensions to the Fast Downward (FD) Planning System (Helmert, 2006). Unlike FF, Fast Downward utilises the MPT representation introduced in Section 2.1.3. The MPT representation is analysed to produce causal graphs which encode information about the dependencies between variables induced by the actions in the domain. Causal graphs are used in this thesis as part of the agent decomposition algorithm presented in Chapter 4.

The idea of causal graphs has persisted in various forms throughout the history of automated planning (Newell and Simon, 1963; Knoblock, 1994). They were previously called dependency graphs (Jonsson and Bäckström, 1995) before coming to be known by their current name (Williams and Nayak, 1997). FD creates separability causal graphs, so called because whenever there is a minimal plan π to change a variable v, there is a connection on the graph from v to every other variable that π could potentially change the value of. In FD, causal graphs are used as part of the causal graph heuristic which is used to guide the search.

Definition Let Π be a multi-valued planning task with variable set V. The *causal graph of* Π, $CG(\Pi)$ is the directed graph with vertex set V containing an arc (v, v') iff $v \neq v'$ and the following condition holds:

- **Transition condition** There is an action (or axiom) that can affect the value of v' which requires a value for v in its precondition (or condition).

Some definitions of causal graphs also include a second edge condition:

- **Co-occurring effects** The set of affected variables in the effect list of some operator includes both v and v'.

The results presented in this thesis are based on the definition that does not include edges based on co-occurring effects along with another small modification (Section 4.1.1). Generating a causal graph is simply a matter of iterating through the actions of a problem and checking to see which edges they induce in the causal graph.

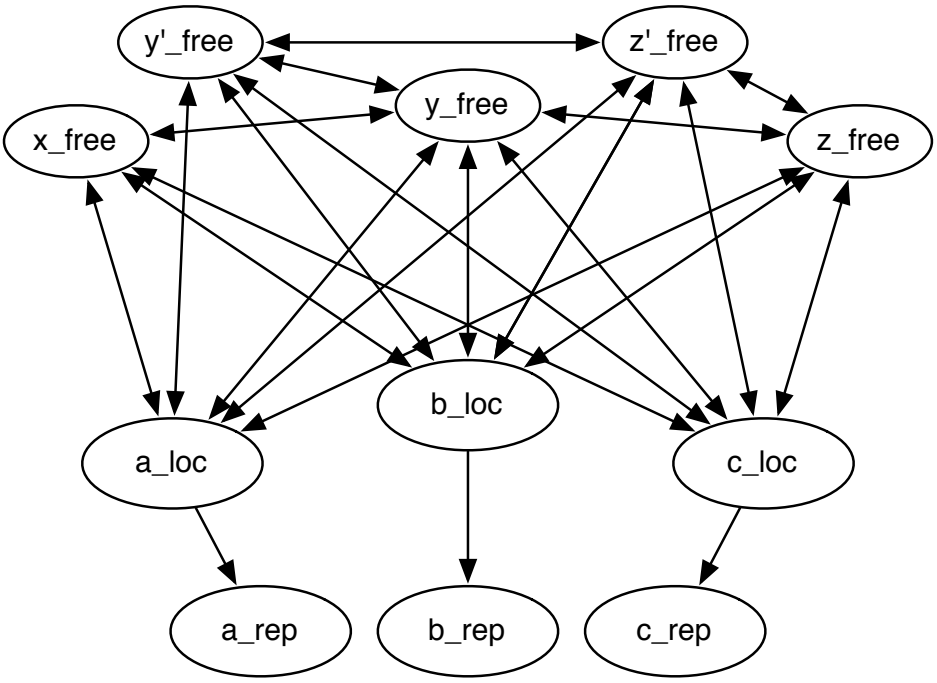

Figure 2.3: The causal graph for the Robots domain shown in Figure 1.1 based on the expected MPT representation.

The causal graph for the Robots example problem from the introduction (Figure 1.1) is shown in Figure 2.3. The causal graph gives a representation of the underlying structure of a domain by showing which variables are dependent on one another. The variables that represent whether or not a particular location is free have a lot of incoming and outgoing edges, whereas the variables for whether or not a robot has reported are only linked to that robot's location variable. While the causal graph shown in Figure 2.3 is of a presentable size, most of the planning problems dealt with in this thesis have causal graphs with thousands of interconnected nodes.

This chapter has so far only discussed single-agent planning work, covering all the methods and ideas that are incorporated into the multiagent planning approaches presented in this thesis. This includes the classical planning formalism, PDDL, the MPT representation, FF's heuristic planning methods and causal graphs. The next part of this chapter discusses the multiagent planning literature as a whole and looks at the complexities involved with introducing multiple agents to the classical planning problem.

2.2 Multiagent Planning

As discussed in the introduction, the multiagent planning literature is diverse, with many loosely connecting strands (de Weerdt et al., 2005; de Weerdt and Clement, 2009). Early work tended to focus on either communication (Georgeff (1983); Grosz and Kraus (1996)) or on the plan coordination problem (Durfee and Lesser, 1991), the latter involving methods for merging existing plans into global plans and dealing with insincere agents (Ephrati and Rosenschein, 1993b, 1994).

In order to bring structure to the research area, there are plans to create a multiagent planning competition similar to the International Planning Competition. The hope is to provide a focal point for research that will drive progress, encourage coordination and facilitate collaboration between projects. However, before a multiagent planning competition can be set up, the exact nature of the problems to include needs to be discussed. A recent talk at the multiagent planning workshop (ICAPS 2013) raised the following questions as possibilities for the type of multiagent planning problem to be covered:

> Observability? Discrete/durative, conditional, hierarchical actions? Interacting actions? Boolean/numeric fluents? Probabilistic or (non)deterministic? Constraints on state-trajectories? Different goals or utilities? Cooperation, competition or teams and coalitions? Common knowledge of the problem?[1]

While this list is not exhaustive, it shows how disconnected the multiagent planning research area currently is. Each paper gives a different collection of answers to the questions posed above. Furthermore, there are no right or wrong answers to those questions, different modelling processes and real-world considerations will lead to different instantiations of the multiagent planning problem.

By choosing just one set of assumptions for the competition, multiagent planning would have a distinct research goal and the structure of the research area would be improved. Different approaches could be contrasted by their proximity to the research area provided by the competition. However, this is a long-term task and, there is no agreement on which assumptions should be made. In the meantime, it is important that multiagent planning work is explicit about its assumptions so that it does not exacerbate the problem. In particular, the community is in need of work that is closely connected with existing, well-defined research areas, in order to provide reference points for the current literature.

[1] Daniel L. Kovacs. Presentation: Some thoughts and ideas on the organization of a multi-agent planning competition. ICAPS 2013 Multiagent Planning Workshop.

In order to achieve this, the approach taken by this thesis is to focus on multiagent *classical* planning; an area of multiagent planning that remains as close to the well-defined single-agent classical planning formalism as possible. In other words, to study finite, fully observable, deterministic, static (environmentally) domains with implicit time, offline planning and multiple agents. While this means that there are many multiagent considerations that are not addressed by this thesis, it provides results that are important for multiagent planning as a whole.

The multiagent planning research most relevant to this thesis is therefore that which remains close to the classical planning problem. The presented work is split into two main strands; the first deals with completely cooperative scenarios where the goal is to find a joint plan between the agents (Boutilier, 1996; Boutilier and Brafman, 2001), the second strand deals with strategic multiagent planning where agents are assumed to be self-interested and have preferences over their plans (Jensen et al., 2001).

The completely cooperative multiagent planning research area involves multiagent decompositions (Nissim et al., 2012), plan synthesis (Brafman and Domshlak, 2008) and planning with concurrent actions (Boutilier and Brafman, 2001). The major sub-problems from strategic multiagent planning are defining solution concepts (Bowling et al., 2002), computing stable solutions and coalition formation (Brafman et al., 2009) and mapping strategic planning onto game theory (Larbi et al., 2007). As the term 'cooperative' is overloaded by its use in game theory, the non-strategic case will be referred to as 'shared-goal' multiagent planning.

2.3 Shared-Goal Multiagent Planning

The phrase 'Shared-Goal Multiagent Planning' is used to describe all types of multiagent planning in which there is a single goal and the objective is to find a plan that achieves the goal without taking into account the amount, or cost, of actions performed by each agent. This is in contrast to Strategic-Multiagent Planning which assumes that agent's have their own goals and/or preferences over possible plans.

As with the introduction to single-agent planning, the first step is to look at planning formalisms. There are three main multiagent planning formalisms in the literature; MAPL (Brenner, 2003), MA-STRIPS (Brafman and Domshlak, 2008) and MA-PDDL (Kovacs, 2012). Out of these, the formalism most closely related to multiagent *classical* planning is MA-STRIPS, which is an extension of STRIPS-planning presented in Section 2.1.1.

2.3.1 MA-STRIPS

The idea behind MA-STRIPS can be traced back to multi-entity models, multiagent models that can be augmented with STRIPS style actions to give a cooperative multiagent planning domain (Moses and Tennenholtz, 1995).

Definition An MA-STRIPS problem for a set of agents $\Phi = \{\phi_1, \ldots, \phi_n\}$ is given by a quadruple $\Pi = \langle P, \{A_i\}_{i=1}^n, I, G \rangle$ where:

1. P is a finite set of atomic formulas of $\mathcal{L}$,
2. $\{A_i\}_{i=1}^n$ is the set of actions for each agent,
3. $I \subseteq P_{\mathcal{G}}$ is the initial state of the problem, and
4. $G \subseteq P_{\mathcal{G}}$ is the goal of the planning problem.

This problem is identical to the standard planning definition given in Section 2.1, except that the set of operators O has been replaced with a set of ground actions for each agent. Each action is a STRIPS-style action of the form $\langle n, pre, eff \rangle$ as in the single-agent definition. Therefore, the problem $\langle P, \bigcup_{i=1 \to n} \{A_i\}, I, G \rangle$ is a single-agent classical planning problem.

The MA-STRIPS formalism for multiagent planning is perhaps the simplest possible extension of the classical planning approach. It does not have any constraints on the different action sets, so a random partitioning of the ground actions in a single-agent planning problem can form an MA-STRIPS problem. Also, MA-STRIPS itself does not commit to a particular form of action execution, it was originally intended to be used with synchronous actions in Moses and Tennenholtz (1995), but is applied to problems with asynchronous actions in Brafman and Domshlak (2008). This is an important point in relation to the need for explicit assumptions in multiagent planning discussed in the previous section.

While there is a clear relation between MA-STRIPS and classical planning, the drawbacks mean that it is not directly applicable to the methods used in this thesis, presented in Chapter 3. Firstly, the algorithms presented in this thesis utilise the MPT representation (Section 2.1.3) which includes more structure than MA-STRIPS. Secondly, Section 3.2 argues that agents should be defined in terms of the variables in a domain, not the actions. Finally, 3.3 argues that in order for parts of the problem to be interpreted as agents, certain conditions must hold between their associated variable sets or induced action sets.

2.3.2 Multiagent Decompositions

While multiagent planning is a growing area, the majority of planning domains are still written in single-agent PDDL. This is the case even for domains based on multiagent scenarios. In order to utilise these domains for multiagent planning, an algorithm is needed that can automatically decompose them into multiple agents.

The idea of decomposing planning domains is not new (Lansky, 1991). However, it is only relatively recently that generalised methods for calculating decompositions based specifically on multiagent structure have been researched. Nissim et al. (2012) is the only other known work to provide automated decompositions that are specifically agent-based and do not require any *a priori* domain knowledge.

The main goal of Nissim et al. (2012) is not decomposition, but to provide pruning methods for optimal planning and one of which is based on the multiagent structure of a planning problem. In a multiagent planning problem, as they define it, if an agent must achieve a particular subgoal on its own, and that subgoal is the next thing to be achieved, then all other agent's actions can be pruned until that subgoal has been achieved. For example, in a house building domain, if the next subgoal is to paint a room, and only the painter agent can achieve that, then only the painter agent's actions need to be considered until the room has been painted. All other actions can be pruned from the search tree until the subgoal is achieved.

As this method requires a multiagent decomposition, and most domains are written as single-agent planning problems with no automated multiagent decomposition methods available, the paper provides an algorithm for computing one. The algorithm utilises the definition of commutative actions (Haslum and Geffner, 2000). *Commutative actions* are actions such that neither one achieves or destroys a precondition of the other. Commutative actions can therefore be exchanged in order (if they occur next to each other in a plan), without affecting the outcome or validity of a plan.

In the paper, the idea behind commutative actions is extended to tunnelling macros and also to a multiagent pruning optimisation which works by focusing on single subgoals at a time. As in MA-STRIPS, they define a decomposition as a partitioning of the problem's action set. Under a given partitioning, the actions are split into public and private actions. Private actions are commutative with all actions in other partitions, all other actions are public. This means that the effect that private actions have on the environment is localised to the partition involving that action.

Private actions have the property that, in an optimal plan, they can always be

followed by another private action from the same partition set. This is because a private action only changes what is possible in terms of other private actions from the same partition set, so, if it is not going to be followed by another action from that partition set, then there was no point in doing it in the first place. This result only works for optimal planning, as it requires the assumption that the previous action added was part of the optimal plan, so this technique is not applicable to the heuristic methods presented in this thesis. However, the decomposition method for creating the multiagent partitioning is of interest.

To create the decompositions, an action graph is defined such that there is an undirected edge between actions a_1 and a_2 if they are not commutative. A symmetry score Γ measures the probability of a sequence containing a private action followed by an action from a different partition appearing during search.

$$\Gamma(\{A\}_{i=1}^{k}) = \sum_{i=1}^{k}(pr(a \in A \wedge private(a)) * pr(a \notin A_i))$$

The probabilities in this equation are calculated as the ratio of actions in the domain with the relevant properties and $private(a)$ evaluates to true only when a is a private action. The idea of the symmetry score is to rank partitions based on the number of actions that are likely to be pruned during search.

This measure of Γ assumes that the probability of each action appearing at any point in the search is equal. Furthermore, finding an action partitioning with the highest symmetry score requires an evaluation of each member of the exponentially large action partition set. The decomposition algorithm presented in the paper utilises unreported heuristic methods for creating a manageable number of partitions and returns the one with the best score.

While this process leads to an action decomposition with properties similar to the one presented in this thesis, it does not explicitly look for multiagent decompositions but instead for decompositions with the lowest amount of interaction between agents. This means that it is less effective at providing decompositions for domains with significant interaction required between the agents. However, it also means that decompositions can be returned for non-multiagent domains that are useful in the sense that they can improve the performance of their optimal planning algorithm. The paper leaves more accurate decomposition finding as an open challenge - one that this thesis contributes to.

2.3.3 Multiagent Plan Synthesis

The preceding has discussed multiagent planning formalisms and methods for generating multiagent planning domains. Once a problem is defined, the obvious next step is to try and solve it. This section discusses the shared-goal multiagent plan synthesis approaches in the literature that are most relevant to this thesis.

While multiagent planning can act as a divide-and-conquer strategy that deals with simple single-agent subproblems, it also creates a coordination problem. This coordination problem grows exponentially with the number of agents and tends to dominate search complexity. Because of this, many multiagent approaches focus on loosely coupled domains.

The idea of exploiting *loose coupling* among agents' plans is developed in Brafman and Domshlak (2008). They investigate how to deal with coordination points in a multiagent planning problem based on a distinction between public and private fluents similar to the public/private actions definition of the previous section. Another approach, Nissim et al's distributed multiagent planning algorithm (Nissim et al., 2010), also exploits loose coupling, solving a distributed CSP for those parts of the global plan where individual agent's contributions overlap.

The degree of coupling in a multiagent planning domain measures the complexity of the coordination planning problem under a multiagent approach. There are two parameters involved in determining the coupling of a system: the number of other agents that each individual agent can affect (by altering preconditions of their actions), and the number of 'interacting' actions that an individual agent must perform to solve the problem. The more interacting actions, and the more connections exist between agents, the tighter the system is coupled.

Brafman and Domshlak (2008) build an agent interaction graph, a digraph that has an edge between two nodes (agents) if an agent can affect another by either creating or destroying a precondition of one of the other agent's public actions. These digraphs give an idea of how coupled a system is. Their algorithm works for domains where the digraph is acyclic, and such domains can be considered as loosely coupled.

Brafman and Domshlak consider loose coupling to be "a natural property of practical multiagent systems." However, the digraphs for all the example domains presented in the introduction (Section 1.2) are maximally connected. This means that they are cyclic, and in that sense those domains are strongly coupled. Most of the domains dealt with in this thesis are loosely coupled. However, the distinction is still useful because, in

general, loosely coupled domains can be solved faster than tightly coupled domains.

Returning to the plan synthesis algorithm, coordination points are defined as the point where an agent executes a public action. The planning problem is then redefined as a constraint satisfaction problem with each variable denoting an agent. Each possible instantiation of a variable is denoted by a public action and a time (in the sequence of actions in the plan) for that action to take place. The constraints over the system are: that the coordination points form a valid sub-plan assuming their internal preconditions are met, and that the internal preconditions can be met in time for each coordination point.

It is shown that time complexity is dependent on the individual planning between coordination points, exponential in the minmax number of agent commitments and exponential in the tree width of the moral graph of the agent interaction graph. Therefore, it is not (directly) exponential on the number of agents. However, adding another agent to most domains would increase the tree-width of the graph, and therefore increase the complexity exponentially. The only way for this not to be the case is if the newly added agent is a root node or leaf node in the agent interaction graph, meaning that it only directly interacts with one other agent.

The results from the previous paper are extended in Nissim et al. (2010), which presents a fully distributed multiagent planning algorithm for shared-goal planning problems described using MA-STRIPS. The research is evaluated using modified versions of problems from the International Planning Competition IPC-2008 International Planning Competition (2008). The particular domains used are those most suitable for conversion to a multiagent problem: Logistics, Rovers and Satellites. These domains are chosen because there is relatively little interaction required between the agents. The Satellites domain is the least coupled while the Logistics domain the most tightly coupled of the three. While the paper only focuses on three of the IPC domains, it shows that they can be solved efficiently with a multiagent approach, which provides motivation for the work presented in this thesis.

In summary, previous multiagent plan synthesis work has focused on loosely coupled domains where the agent interaction graphs are acyclic and shown that these domains can benefit from a multiagent approach. In contrast, this thesis looks at all domains that have an inherent multiagent structure and, while this structure imposes conditions on the interaction between agents, it covers both tightly and loosely coupled domains with cyclic or acyclic agent interaction graphs.

It should also be noted that the work presented in this section, as with many

multiagent approaches, used single-agent defined planning problems with a hand-coded decomposition. This hand-coding process can be time consuming and requires familiarity with the PDDL representation of a domain. This process is automated by the agent decomposition algorithm presented in Chapter 4.

2.3.4 Concurrent Actions

Even though concurrent actions break the classical planning assumption of sequential actions, the ability to perform them is fundamental to multiagent systems in general. They are discussed in this section as an extension to the classical multiagent planning problem. Concurrent actions are also dealt with in the temporal planning community, however, discussion of this area is not pursued in this thesis.

In planning, concurrent action interactions can be either positive or negative. An example positive concurrent interaction would be two agents simultaneously lifting a table whilst keeping it level so that an object placed on top will not slide off. An example of a negative interaction would be that multiple agents cannot pass through a small doorway at the same time (see Chapter 6.2).

The work on concurrent actions most closely linked to the classical multiagent planning problem is that presented in Boutilier and Brafman (2001). They show how STRIPS representations of actions can be modified to include a concurrent action list that describes the restrictions on the actions that can (or cannot) be executed concurrently in order to have a specified effect.

For a domain with n agents, a joint action is comprised of an action for each agent in the domain. This means that the joint action space is exponential in the number of agents. For agents $\Phi = \{\phi_1, \ldots, \phi_n\}$, with action sets $A_1, \ldots, A_n$, the joint action space is $A_1 \times \ldots \times A_n$. Every possible combination of actions from each agent exists in the joint action space. It is clearly not feasible to define the effects of joint actions individually. Even in very small problem instances, for example just 5 agents with 3 actions each, there are 243 possible joint actions. The method proposed by Boutilier and Brafman (2001) is to add concurrent action constraints to the action definitions in PDDL. These allow for the full joint action space to be defined without having to define each possible joint action individually.

To achieve this, each action in the multiagent domain must be associated with some agent. The parameters of all actions are modified to include an agent variable that has been artificially added to the domain. A concurrent action constraint lists all other

actions that either must or must not appear in a joint action at the same time as the action being defined. These constraints may contain variables in the same way that operator definitions in PDDL can contain variables and they can also contain the $=$ relation which holds when its two arguments refer to the same object. For example, in the table lifting scenario, a concurrent action constraint would specify that another agent must perform the table-lift action at the same time.

A concurrent action constraint that includes action *a* means that *a* must appear in the same joint action, otherwise nothing will happen. A concurrent constraint that includes the negation of *a* means that *a* must not appear in the same joint action. For example, the move action with a concurrent action constraint to prohibit other agents from simultaneously moving into the same space could be defined as follows:

```
(:action move
:parameters (?a - agent ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (free ?y) (connected ?x ?y))
:concurrent (not (and (move ?a2 ?z ?y) (not (= ?a1 ?a2))))
:effect (and (at ?a ?y) (not (at ?a ?x)) (free ?x) (not (free ?y)))
)
```

This says that the action must not occur at the same time that another agent moves from any location `?z` to the same destination location `?y`.

The benefits of this approach are that the full joint action space can be defined more efficiently than by looking at each possible joint action individually. However, in a multiagent planning problem, this type of definition requires knowledge of the interactions between every possible action in the domain. If a new agent is to be added to the domain with its own actions, then, first, every other action in the domain must be checked and then new concurrent constraints must be written. This thesis introduces a method for defining concurrent action constraints based on the objects in the domain. This means that should a new agent be added, the way that it can interact with the environment is defined by the concurrent action conditions specified on the environment, and it does not need knowledge of every other action in the domain.

The paper continues to propose a partial-order planning algorithm that can deal with concurrent action constraints which extends earlier work such as that by Knoblock (1994). The algorithm is shown to be sound and complete, but its empirical qualities are not discussed. Planning with concurrent actions remains an open challenge for multiagent planning research, this is discussed further in Chapter 6.

2.4 Game-Theoretic Concepts for Planning

While the majority of multiagent planning assumes shared goals, there are many approaches that focus on the possible strategic elements of a multiagent environment. Parsons and Wooldridge (2002) point out that, although there is a large body of research in cooperative multiagent domains:

> It has come to be recognised that in fact, benevolence is the exception; self-interest is the norm.

This section introduces the game-theoretic concepts that are relevant to multiagent planning.

Game Theory (Gibbons, 1992; Osborne and Rubinstein, 1994) has become increasingly prevalent in multiagent systems research in recent years, and is now seen as an "important theoretical basis" (Brafman et al., 2009) of multiagent research. There is even a textbook (Shoham and Leyton-Brown, 2008) devoted to the area where these two disciplines meet.

Game theory entered multiagent systems research in 1985 (Rosenschein and Genesereth, 1985; Rosenschein, 1986) but has its roots in von Neumann and Morgenstern's work (Morgenstern and Von Neumann, 1944). It deals with the interactions between self-interested agents which are attempting to maximise their own utility. In game theory, it is generally assumed that each agent is rational, and the rationality of all other agents is common knowledge. By making this assumption, it becomes possible to reason about other agents' behaviour, and therefore to make predictions about an otherwise chaotic multiagent domain.

The basic problem formalism in game theory is the normal-form game, though there are of course many different variations and extensions of this problem. A normal-form game with n players is represented by $\Pi = \langle N, S, \{u_i\}_{i=1}^{i=n} \rangle$. Where, $N = \{1, \ldots, n\}$ is the set of players, $S = S_1 \times \ldots \times S_n$ such that each S_i is a set of strategies for player i, and $u_i : S \to \Re$ describes the payoff to player i under each possible combination of strategies in S.

It is assumed that a player i can choose to play a strategy $s \in S_i$ or a mixed strategy made up of a probability distribution over all the strategies in s_i. If we have a profile of mixed strategies $x = (x_1, \ldots, x_n)$ then let x_{-i} denote the strategy of everyone except player i, in other words:

$$(x_1, \ldots, x_{i-1}, \ , x_{i+1}, \ldots, x_n)$$

and $(x_{-i}; y_i)$ denote the strategy

$$(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_n)$$

where y_i has replaced x_i in x.

A mixed strategy y_i^* is a *best response* for player i to x_{-i} iff for all possible strategies y_i, $u_i(x_{-i}; y_i^*) \geq u_i(x_{-i}; y_i)$. In other words, if i knows what the other players (agents) are going to do, then its best response is one of the strategies that maximises its own utility.

It is now possible to introduce the concept of equilibria in games. Informally:

> An equilibrium is a *joint* solution for all the agents, such that there is no reason for any agent to change their own choice of actions given their desire to maximize some real-valued utility function. (Bowling et al., 2002)

The most famous equilibrium concept is the Nash Equilibrium (Nash, 1951). A strategy profile x is a *Nash Equilibrium* iff for every player i, x_i is a best response to x_{-i}. In other words, a Nash Equilibrium is a mixed strategy where no player can benefit by unilaterally deviating from their strategy. An important result from game theory is that every finite normal form game has a Nash Equilibrium (Nash (1951)). The concept of Nash Equilibrium is applied to a multiagent planning in Bowling et al. (2002) and Larbi et al. (2007), which will be discussed towards the end of this chapter.

2.5 Strategic Multiagent Planning

This section looks at the application of game-theoretic concepts to multiagent planning. As with previous sections, the first part introduces strategic multiagent planning formalisms from the literature.

2.5.1 Coalition-Planning Games

Coalition-Planning Games (CoPGs) appear in Brafman et al. (2009) as an extension of MA-STRIPS to the strategic case. In a CoPG, with $\Phi = \{\phi_1, \ldots \phi_n\}$ being the set of agents, the domain is represented by a 6-tuple $\Pi = \langle P, A, I, G, c, r \rangle$. P and I are as in MA-STRIPS with $A = \{A_i\}_{i=1}^k$ comprising of a set of actions for each agent in the domain. As a departure from MA-STRIPS, $G = \{G_i\}_{i=1}^k$ contains a goal state for each agent, $c : A \to \Re^+$ is a cost function representing the cost of each action and $r : \Phi \to \Re$ is a reward function giving the reward each agent gets for reaching its goal state.

Brafman et al. (2009) are interested in the complexity of MAP problems that have a specific interaction structure. In the paper they show "that when a certain graphical structure induced by the system is acyclic, stable plans can be found in time polynomial in the description size of the MA system." This acyclic nature is the same as discussed in Section 2.3.3.

As with MA-STRIPS, CoPGs are the natural extension of classical planning to a strategic multiagent setting. The additional requirements are a distribution of the goals between the agents and a way of formulating the preferences that agents have over plans.

Following the CoPG definition, the reward of agent φ for plan π is $r(\varphi)$ if π achieves g_φ and zero otherwise. This follows the simple goals assumption to the extent that it does not allow for agents to have rewards for partial completion of plans. However, goals may have different rewards and actions may have different associated costs. An agent can always guarantee zero payoff by doing nothing, i.e. not receiving a reward but not incurring a cost. The utility of an agent's plan $u_\phi(\pi)$ is defined as the reward (if the goal is completed) minus the sum of the cost of the actions taken.

Given that agents have preferences over possible plans the natural next question is: Can we calculate a global plan that is acceptable to all agents? To do this we need an equilibrium concept for multiagent planning.

2.5.2 Equilibrium Concepts for Strategic Multiagent Planning

There are two approaches in the literature for applying equilibrium concepts to multiagent planning. The first is to convert planning problems into games, so that equilibrium concepts from game theory can be directly applied. The second is to define new equilibrium concepts that can be directly applied to planning problems. The following papers look at a way to convert multiagent planning domains to games. The first deals with multiagent planning with asynchronous actions while the second looks at the synchronous action case.

2.5.2.1 Mapping Planning Problems to Games

Larbi et al. (2007) outline an approach for solving multiagent planning problems with game-theoretic methods. It assumed that, in general, the order of execution and interleaving of each agent's plan is not known in advance, so that each agent has to plan for all possible interleavings. However, when agents coordinate to build a common

plan, the uncertainty of the execution is removed.

A game is associated to a multiagent planning problem, so that the 'best' plan for an agent can be found by employing the notion of Nash Equilibrium. They represent an agent by a triple $\langle A^i, \Pi^i, G^i \rangle$ where A^i is the set of agent i's actions, Π^i is the set of agent i's possible plans[2] and G^i is a set of (possibly multiple) goal states.

Shuffle sets are introduced, written as $p_i \oplus p_j$, which contain every possible ordering of the two plans p_i and p_j while preserving the order of actions within the original plans. So, for example the plans $[a_1, a_2]$ and $[b_1, b_2]$ have shuffle set:

$$\{[a_1, a_2, b_1, b_2], [a_1, b_1, a_2, b_2], [a_1, b_1, b_2, a_2],$$

$$[b_1, a_1, b_2, a_2], [b_1, a_1, a_2, b_2], [b_1, b_2, a_1, a_2]\}.$$

An agent's plan is then evaluated against all possible final states given by all possible elements of the shuffle set.

By considering the possible outcomes of shuffle states in terms of the agents' goals, a preference over possible shuffle sets is given. An agent prefers a shuffle set where it is *always satisfied*, such that there is no element of the set where the agent does not reach its goal. Next, an agent will want *mutual interest*, so that if the agents coordinate they can choose a particular element of the shuffle set where they all meet their goals. After that, an agent will hope for *dependence*, meaning that they can achieve their goal with coordination from the other agent and it does not hurt the other agent to give this help. The next possible state is *antagonism*, the agents cannot be jointly satisfied. Finally, there may be situations where the agent is *always dissatisfied* and cannot ever reach its goal.

Given a plan for each agent, the shuffle set can be calculated and analysed so that the preference of an agent for that particular shuffle set can be attained. A game is obtained from the MAP problem by associating each possible plan $\pi \in \Pi^i$ to a strategy and then assigning utilities based on the plan classification above worked out from the shuffle sets. The game has an associated Nash Equilibrium which is a joint plan such that no agent can unilaterally deviate to a better shuffle set category.

Given the setup, the categories given are a sensible grouping of possible situations. Unfortunately, there is no further work in the area, as other papers either focus on synchronous actions or asynchronous actions without the notion of shuffle sets. Shuffle

[2] Which may be smaller than the full set of possible plans due to computational limitations of the agent. However, it is assumed to be closed under the subplan operation (i.e. if plan π is in Π^i then all subplans of π are also in Π^i)

sets are an interesting idea; however, it is hard to motivate the need for a domain where action execution is unpredictable to such an extent. The fact that the ordering of each agent's individual plan is preserved is of little consolation, as the most interesting multiagent planning scenarios are ones where agents rely on other agents to fulfil the preconditions of their actions.

The *always satisfied* category is only ever met if the agent can achieve its goal in the environment by itself. This is easy to see by considering the shuffle set where the agent's plan is executed entirely before any other agent executes an action, and when no other agent is choosing a plan that removes a precondition needed by the original agent.

The advantage of the model proposed by the paper is that given the calculation, Nash Equilibria can be directly applied to the game to find a stable solution. However, the main problem is that the construction of the game requires enumeration of every possible plan for each agent. Not only does each agent's full list of plans need to be calculated, and each plan evaluated against every possible combination of each other agents possible plans, but, for each evaluation, every member of the shuffle set has to be calculated. This is infeasible for all but the smallest planning problems with two agents and very few actions.

Bowling et al. (2002) are concerned with defining equilibria for the synchronous action case and is one of the first attempts to define equilibria for multiagent planning using notions from game theory. They define the transition relation of the domain to be $R \subseteq S \times A \times S$, where S is a set of states and $A = \{A_i\}_{i=1}^{k}$ from the CoPG definition. In other words, actions are assumed to be performed synchronously with the outcome depending on each action performed. They also assume that goal states won't necessarily correlate, and, at the end of a planning run, maybe only one agent will have been able to achieve its goal. A further classical planning assumption that is dropped is that there may be multiple initial states for the system and the agent may not know which one the system is in.

They define a solution in terms of a set of actions to perform in each possible state of the system, instead of a solution being a string of actions to perform. For a solution π, they categorise the strength of the solution as follows:

1. π is a *weak* solution for agent i iff for any possible initial state it is possible to get to a goal state of i's.

2. π is a strong *cyclic* solution for agent i iff from any state that it is possible to reach, it is possible to get to a goal state of i's.

3. π is a strong solution for agent i iff all possible execution paths contain a goal state of i's.

4. π is a perfect solution for agent i iff all possible execution paths reach and maintain a goal in a finite amount of steps.

The list is very similar to the categories defined in Larbi et al. (2007), the further down the list, the better the solution. A plan π is an equilibrium solution iff for all agents i the strength of solution π for i is the strongest it can be, given the actions each other agent is performing in π. This closely mirrors the definition of Nash Equilibria in Game Theory. However, in contrast with Nash Equilibria, it is not necessarily true that an equilibria exists.

In order to check if a solution is an equilibrium, it is necessary to check the strength of every other possible plan of each agent in the system. This is similar to the calculation required in Larbi et al. (2007), except that instead of determining the outcome for each element of the shuffle set, each possible execution trace through the system for each possible starting state needs to be found.

The two papers in this section successfully define equilibria for MAP domains. However, they both have the drawback that using the equilibria requires far too expensive a calculation to convert the multiagent planning problem into a game. This suggests that the correct path is to define equilibria directly for planning problems as is discussed in the next section.

2.5.3 Defining a solution concept for multiagent planning

A solution concept in a multiagent planning domain is not easy to define without the expensive conversions to games used in some of the previous papers. Brafman et al. (2009) point out that "Intuitively, a solution is *stable* if there exists no set of agents, all of which can increase their utility by jointly adopting a different plan." But the exact definition of this is problematic.

The definition they adopt is that a solution π for a coalition-planning game Π of agents Φ is *stable* iff there is no alternative plan π' involving a subset of agents $\phi' \subseteq \Phi$ such that $u_\varphi(\pi') > u_\varphi(\pi)$ for all $\varphi \in \Phi'$. In other words, if a group of one or more agents could come up with another plan by themselves which is strictly better for each agent concerned, then the solution is not stable. This is different to the situation where

one agent must be strictly better off while the other agents have to do at least as well because each agent must have sufficient motivation for changing plan.

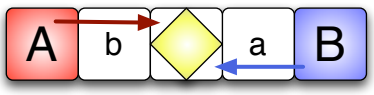

Figure 2.4: The agents cooperate to maximise utility

Consider Figure 2.4, an example of a parcel delivery CoPG which has the same setup as the example problem from the introduction except that each action has cost 1 and the goals are split up between the agents. Agent *A* has to deliver parcel *a* to the depot (yellow diamond) while agent *B* is trying to deliver parcel *b* to the same place.

It should be obvious that, given the ability to make deals, the agents are best off delivering each other's parcels. However, imagine that we have this joint plan and want to check if it is stable. Agent *A* will want to deviate to the plan where it does nothing, reasoning that its parcel will be delivered anyway and it can increase its utility by not helping deliver agent B's. However, this reasoning is obviously flawed, as agent B will no longer have a reason to help agent A and will not deliver its parcel. Therefore, in Brafman et al. (2009), when considering a possible deviation, the agents not in the deviating subset are assumed to do nothing, this way they will not be helping without reason.

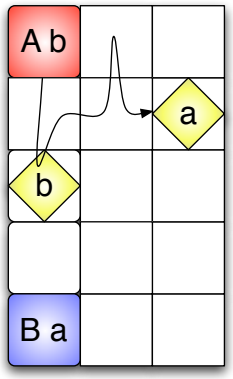

Figure 2.5: Agent A has a stable plan that includes a useless utility costing deviation

However, there may be situations where the non-deviating agents still have reason to help the deviating ones. This means that the definition of stability given in Brafman et al. (2009) has some rather unintuitive results. For example, consider the joint plan where agent A follows the indicated path in figure 2.5, picking up and delivering agent

Figure 2.6: Agent B controls the bridge

B's parcel (while B brings parcel *a* up to depot *b*) and then delivering its own via an obviously suboptimal route. This is a stable solution even though it seems intuitively obvious that agent A has a more direct route to depot *a* once it has picked up its parcel. This is because, by assuming that the agents do nothing when not deviating, the implicit deal between A and B is broken even though A is still upholding its end of the deal. Assuming no action for non-deviating agents achieves the intended goal of cancelling deals with deviating agents; however, it does not take into account deals that the deviating agents continue to abide by. *A* and *B* cannot jointly deviate to the improved plan because, according to the definition, the plan deviated to has to be strictly better for both agents.

Furthermore, assuming that non-deviating agents do nothing prevents them from being able to perform any destructive actions if the other agent deviates from a deal. In Figure 2.6, the black line represents a bridge that can only be crossed once. Agent B will take his parcel across the bridge, thereby breaking it; agent A is then unable to deliver its parcel. However, given coalitions, the optimal solution is for agent A to deliver both parcels. Agent B has increased utility as it no longer has to spend the cost of crossing the bridge and agent A has increased utility because it manages to achieve its goal. Therefore this solution strictly dominates the one where agent *B* delivers its own parcel and *A* does nothing.

If it is assumed that agents not in the deviating subset simply do nothing, then agent A will want to deviate from the optimal solution to one where it does not pick up B's parcel. This violates the implicit deal between them that if Agent A delivers B's parcel, then agent B will refrain from crossing the bridge. The problem of defining a suitable solution concept for multiagent planning is clearly complex and will be covered, in

detail, in Chapter 6.1. The next section looks at plan synthesis for strategic multiagent planning.

2.5.4 Synthesising plans with self-interested agents.

The paper discussed in the previous section also presents results on the complexity of multiagent planning problems that have a specific interaction structure. It was shown "that when a certain graphical structure induced by the system is acyclic, stable plans can be found in time polynomial in the description size of the MA system." This is the same acyclic property that has occurred in many places in the multiagent planning literature, yet does not hold in any of the example domains from the introduction.

The algorithm for finding a stable strategy requires an acyclic agent interaction graph. The interaction graph is isomorphic to the digraphs introduced in Section 2.3.3. The nodes are agents and the edges represent interactions between possible actions, with an edge being present between agent a and agent b if one of a's actions adds or removes a precondition of one of b's actions.

Consider the interaction graph represented as a tree, with each agent's possible strategies on their respective node. A strategy is a sequence of actions that reach the goal state assuming that the agent is supplied with the preconditions it needs from agents further down the tree. Starting at the bottom of the tree, each agent keeps only its strategies that produce the maximum utility (if completed) that are still viable given the deletions already made by agents further down the tree. If, from the removal of strategies, there is no strategy further down the tree that will give an agent a particular precondition, then all subsequent strategies that required it are removed. Once the top of the tree is reached, each agent (from top to bottom) picks a strategy from those remaining in its list. The combined joint strategy forms a stable solution.

Another approach to plan synthesis in strategic multiagent planning is considered in Jonsson and Rovatsos (2011). This approach looks at domains with concurrent actions and assume the existence of an admissibility function that indicates which joint actions are possible. They use this admissibility function as a method for quickly checking whether a action is possible for an agent, given each other agent's plans. This leads to a best-response approach to planning that can find solutions to a certain class of planning problems called congestion games, and performs reasonably well in other, non-congestion game domains.

The best-response approach, given a starting plan, takes each agent in turn and finds

there best response to the current joint plan. The planning problem can be rewritten to be fixed by every other agents' plans and then a standard single-agent planner used to find the best response of the an agent. While, this does not lead to a stable solution in the general case, Jonsson and Rovatsos (2011) show which class of domains a stable solution can be found for. The drawbacks of the approach are that it assumes a reasonable starting plan, which is a joint plan for all the agents, and therefore can be relatively hard to compute, and that it requires an explicit encoding of all concurrency constraints in terms of the admissibility function, and that this function is not integrated in the PDDL specification of the domain.

2.6 Summary

This chapter has surveyed the related literature and introduced the relevant background from single-agent classical planning formalisms to solution concepts for strategic multiagent planning. The classical planning assumptions were discussed and it was argued that it is desirable to keep a multiagent planning formalism as close to these as possible.

The shared-goal multiagent planning work has shown how a multiagent planning approach can be useful for solving standard planning problems. However, there was a gap in the literature for a well-grounded mechanism for computing multiagent decompositions without using human *a priori* domain specific knowledge. Chapter 4 provides such an automated domain independent method. There was also a gap for multiagent planning algorithms that can compete with current state-of-the-art single-agent planners. Chapter 4 also introduces a multiagent planning algorithm that has improved performance (in terms of planning time) over competing single-agent planners on multiagent domains.

It was shown that the strategic multiagent planning literature contains methods for converted planning problems to game theory so that equilibrium concepts can be used directly. However, it was argued that there is no computationally efficient way to do this and that it is necessary to define solution concepts that apply directly to the multiagent planning problems. It was also shown that defining equilibria without transformation is difficult because there is no simple way of defining dependence between agents' actions. Chapter 6 introduces a solution concept that improves on that in the literature and discusses this in more detail. The next chapter provides the multiagent planning formalism that will be used throughout the thesis.

Chapter 3

Multiagent Multi-Valued Planning Tasks

The previous chapters presented and motivated the hypotheses of this thesis, introduced some relevant background information and discussed the related literature. It was shown that there is no unified multiagent planning formalism that has become standard across multiagent planning. This is because there are many different strands of multiagent planning research and each one requires a different formalism suited to its assumptions. Furthermore, the most closely related formalism MA-STRIPS, is not expressive enough for the algorithms presented in this thesis. The formalism presented in this chapter is an extension of the single-agent MPT representation (Section 2.1.3) to the multiagent case (MMPTs). MMPTs are defined in such a way that their agent decompositions have certain properties that are shown empirically in Chapter 5 to correspond strongly with multiagent decompositions as expected from a human perspective.

The first section of this chapter discusses the general properties that an MPT-based multiagent planning representation should have, focussing on what it means to be an agent. Section 3.2 shows how a decomposition of the variable set of an MPT can be used to define the agents in a domain. Actions are discussed in Section 3.3, and it is shown how a variable decomposition completely determines the breakdown and dependencies between actions in a domain, culminating in the definition of MMPTs in the final section of the chapter.

3.1 Agents in MPTs

Recall the definition of an MPT (from Section 2.1.3):

Multi-valued planning tasks (MPTs) A *multi-valued planning task (MPT)* is a 5-tuple $\Pi = \langle V, I, G, X, A \rangle$ where:

- V is a finite set of state variables v, each with an associated finite domain D_v,
- I is a full variable assignment over V called the initial state,
- G is a partial variable assignment over V called the goal,
- X is a finite set of (MPT) axioms over V, and
- A is a finite set of (MPT) actions over V.

An MPT is clearly designed for single-agent planning with a single set each of variables, goals, and actions. The following discusses how each part of the MPT relates to multiagent planning separately.

Variables: A variable is a collection of facts such that only one can be true at a time during execution of a plan. For example, the facts `open(door)` and `closed(door)` can form the possible values of a variable that represents the state of a door. This interpretation of variables as representing the states of parts of the domain suggests that they are suitable building blocks for defining agents. A collection of variables can represent the internal state of an agent which will turn out to be a key component of the approach taken in this thesis.

Initial State: The initial state of a planning problem is independent of the number, or breakdown, of agents in the domain. However, certain parts of the initial state may only be directly relevant to a subset of the agents. If agents are defined in terms of the variable set, then, as I is a full variable assignment with domain V, it is easy to determine which parts of the initial state are directly relevant to each agent.

Goals: While the goal state does not change with the introduction of agents, it may be that different agents want to achieve different parts of the goal. Some goals may be achievable by all agents, and some may be achievable by only a subset of the agents. In strategic multiagent planning, each agent may have its own subset of goals that it needs to achieve and may or may not care if other agents achieve their goals or not.

Axioms: The axioms in the domain are independent of the multiagent decomposition. However, as with the initial state it may be that some axioms are only relevant to certain agents.

Actions: Actions can belong to either a single agent, a subset of agents or all agents in the domain. There are many possible ways that actions can interact or be split

amongst the agents. The type of interactions between the actions determine how difficult it is to solve a domain using a multiagent approach. As with the other elements of MPTs, it is possible to determine the breakdown of actions from a variable decomposition.

The background section showed that the common method for defining a multiagent decomposition is to focus on the breakdown of the actions in the domain. While the distribution of actions is clearly important, it is possible to take a step back and define a decomposition in terms of the variables in the domain. A partitioning of the variables, in turn, directly induces a partitioning of the actions.

The quality of a particular multiagent representation can be measured by a number of factors dependent on the motivation for using multiagent planning in the first place. If the goal is to model real-world multiagent problems as easily as possible, then how easy it is to define a domain will be an important factor. On the other hand, a multiagent approach could be employed simply to try and improve planning times over single-agent planning methods. This thesis focusses on the latter case, in which the complexity of the created subproblems along with the coordination problem are the most important factors.

If a single-agent planning problem can be thought of as a single problem, then a multiagent planning problem with n agents is either $n+1$ or 2^n-1 problems. It is $n+1$ problems if the coordination problem is treated as a whole meaning that there is one problem per agent and one coordination problem. However, if a different coordination problem is defined for each possible subset of agents, then there are 2^n-1 problems in total. As is to be expected, the collected coordination problem, whichever way it is treated, usually dominates the individual agents sub-problems in complexity, especially as the number of agents increases. However, there are many cases where the benefits of splitting a problem into separate agents outweighs the cost of introducing the coordination problem between the agents.

The rest of this chapter outlines an approach that leads to agent decompositions that have intuitively defined agents and coordination problems of relatively low complexity. The next section returns to the Robots domain from the introduction (Section 1.2.1) and discusses the possible agent decompositions of this domain with particular focus on agents' internal states. This example will be used throughout this chapter to explain and clarify the approach.

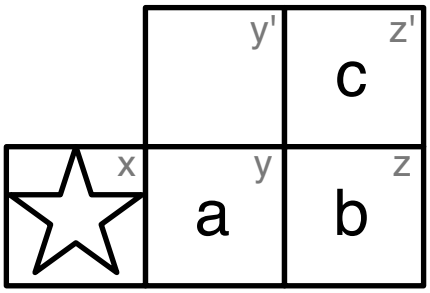

Figure 3.1: An example problem. The larger lower case letters represent robots that need to report to the starred square. The smaller letters represent names for the locations in the problem.

```
(:action move
:parameters (?r - robot ?x - loc ?y - loc)
:precondition (and (at ?r ?x) (free ?y) (connected ?x ?y))
:effect (and (at ?r ?y) (not (at ?r ?x)) (free ?x) (not (free ?y)))
)
(:action report
:parameters (?r - robot ?x - loc)
:precondition (and (at ?r ?x))
:effect (reported ?r ?x)
)
```

Figure 3.2: PDDL representation of the actions in the robot domain.

3.1.1 Example: Toy Robot Domain - Agents

Section 1.2.1 introduced a grid world domain containing multiple robots. The robots can `move` to adjacent grid squares (provided there is no other robot in that space) and `report` at their intended destination which achieves their part of the global goal. The particular problem instance under discussion is reproduced in Figure 3.1 and the operators for this domain are shown in PDDL (using types) in Figure 3.2. The goal for the problem instance shown in Figure 3.1 is

$$\texttt{reported(a)} \wedge \texttt{reported(b)} \wedge \texttt{reported(c)}.$$

As a point of reference, this problem instance contains just 45 ground actions with achievable preconditions.

The intuitive human method for solving this problem is to treat the robot's as separate entities. The process by which a human would find a solution to the problem

was discussed in Section 1.2.1. It makes sense to consider the robots as separate entities, with their own set of `move` actions and their own individual `report` action. However, it is not immediately obvious which generalisable qualities of the problem exist that give rise to such an obvious decomposition.

At first glance, the typing of the domain is a key factor as the type `robot` corresponds directly to the agents in the domain. However, a similar argument could conclude that the locations in the domain are separate agents. Furthermore, associating agents with types presupposes a particular, typed PDDL representation. It is possible to write this domain without types and it is possible to use types but not have them correspond to the robots. There is a much more fundamental property of the domain that leads us to its natural representation in terms of agents. This is explained by revisiting the definition of the term 'agent' itself. While there is no agreed upon definition of an agent, it is commonly accepted that autonomy is a fundamental property.

Perhaps, the most widely used definition is from (Wooldridge, 2001):

> An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

The key point here is that an agent must have *autonomy* over something, for if an agent has no autonomy, then how can it be considered a separate agent?

Relating this to the example problem, the robots have autonomy over their locations and whether or not they have reported. These are facts that, under the intuitive human decomposition, only the robots themselves can change. A robot's location `(at a x)` can only be changed by the action `(move a x y)` and this is an action that it is natural to assume only robot *a* can perform. The location of a robot and whether or not it has reported are an *internal state* of that robot.

Combining the previous discussion of variables with the idea that an agent has an internal state that it has autonomy over, leads us to the following informal definition of an agent in a multiagent MPT:

Informal Notion of an Agent An agent in a multiagent MPT has an associated set of variables that represent its *internal state*.

This property generalises across planning domains and its existence is visible in the majority of cases which have a natural decomposition. It will be shown later that an algorithm based on this informal notion can be used to find the expected decomposition over all the classical planning domains used in the International Planning Competition.

The previous two sections argued that it is the variables of an MPT that are important in defining the agents present in the domain and that, furthermore, these variables should be used to define internal states of agents. The next section formalises this idea and defines variable decompositions of MPTs.

3.2 Variables

The previous section argued that agents can be understood as entities with autonomy over some part of the planning problem and that the part of the planning problem that they have autonomy over represents their internal state. It is natural to assume that this internal state cannot be modified by the actions of other agents. This section discusses how variables can be partitioned to define the internal states of agents in multiagent planning problems.

The definition of an agent provided in the previous section emphasised autonomy, but also included that the agent is "situated in some environment". This is an important point that is sometimes overlooked by multiagent planning approaches (especially those that categorise agents at the level of actions and therefore do not take into account the underlying structure of the domain). There are parts of the planning problem that will not belong to any agent and, instead, form the environment that the agents are acting in.

Figure 3.3 shows how the approach taken in this thesis differs from that in the literature. The standard approach, shown on the left of the figure, is to define agents by partitioning the action set. The agent interaction graph, that contains an edge between agents if they have an action that can affect the possibility of actions performed by the other agent, can, in the worst case, be maximally connected. On the other hand, the approach taken in this thesis is to define agents by internal variables that interact with some environment. In this case, there are no direct links between the agents, all interaction occurs through the environment. In other words, it is impossible for an agent to perform an action that changes the internal state of another agent.

The Robots example from the previous section showed that the agent's locations and whether or not they have reported can be thought of as internal states of the agents. This leaves whether or not a location is free as a property of the environment. In other words, it is not internal to any of the agents.

Variable decomposition A *variable decomposition* of an MPT $\Pi = \langle V, I, G, X, A \rangle$ is a set $\Phi = \{\phi_1, \ldots, \phi_n\}$ along with a set $P = V \setminus \bigcup \Phi$ such that either:

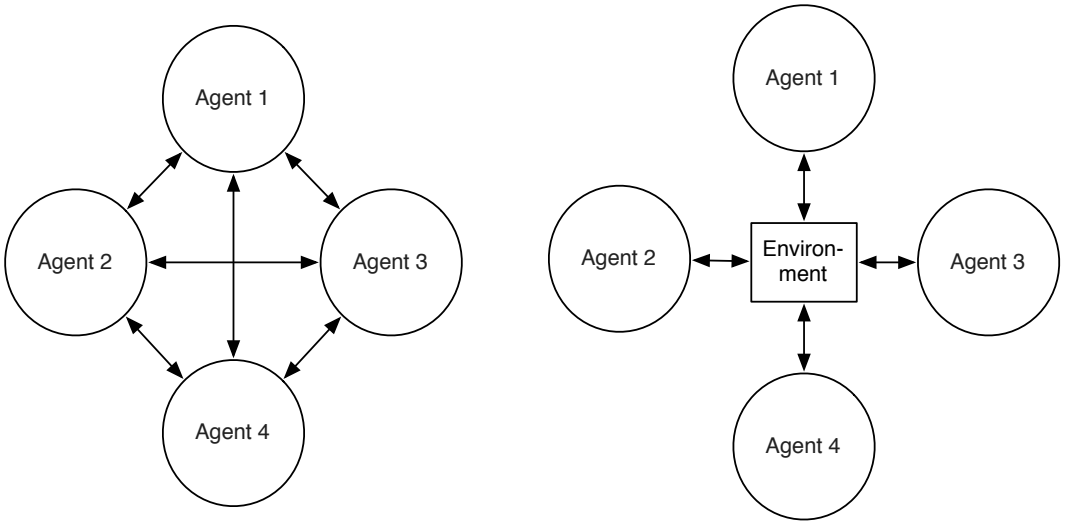

Figure 3.3: Diagram of the two different approaches to multiagent decomposition. The approach from the literature (left) partitions the action set and each agent may interact directly with the others. The approach on the right creates an environment and separate agent variables so that agents only interact indirectly through the environment.

- Φ is a *partition* of V (and $P = \emptyset$), or
- $\Phi \cup \{P\}$ is a *partition* of V.

The set $\Phi = \{\phi_1, \ldots, \phi_n\}$ is used to represent the n agents in the decomposition. ϕ_i or just i is used to refer to agent i. A variable decomposition can be described by Φ only as P can be deduced given knowledge of V.

Informally, a variable decomposition of an MPT assigns a set of variables to each agent and a set of public (or environment) variables such that all variables are covered and each agent's variables are unique to that agent. This definition says nothing about the properties of the decompositions, it does not yet guarantee that the variables represent internal states. To do this, the way in which a variable decomposition induces dependencies between the actions in a domain needs to be discussed. However, first, the variables and their possible decompositions from the Robots example are analysed.

3.2.1 Example: Toy Robot Domain - Variables

The variables that Fast Downward (Helmert, 2006) finds for the Robots problem are shown in Figure 3.4. The initial state shown in Figure 3.1 can be represented in the variable decomposition as:

```
//A variable for whether each location is free or not.
V1 = {⊥, (free x)}
V2 = {⊥, (free y)}
V3 = {⊥, (free z)}
V4 = {⊥, (free y')}
V5 = {⊥, (free z')}
//A variable for the location of each agent
V6 = {(at a x), (at a y), ..., (at a z')}
V7 = {(at b x), (at b y), ..., (at b z')}
V8 = {(at c x), (at c y), ..., (at c z')}
//A variable for whether each agent has reported or not
V9 = {⊥, (reported a)}
V10 = {⊥, (reported b)}
V11 = {⊥, (reported c)}
```

Figure 3.4: The set of state variables for the toy robot domain shown in Figure 3.1. Each variable may only have one value at a time. A state is an assignment of a value to each variable.

V1 = (free x), V2 = ⊥, V3 = ⊥, V4 = (free y'), V5 = ⊥
V6 = (at a y), V7 = (at b z), V8 = (at c z')
V9 = ⊥, V10 = ⊥, V11 = ⊥

The goal state is represented by the partial variable assignment:

V9 = (reported a), V10 = (reported b), V11 = (reported c)

Of particular interest are the variables V6–V8 that collect together the possible locations at which a robot can be found. A sensible decomposition (and the intuitive decomposition from a human perspective) would be to split up the variables so that:

$$\Phi = \{\{V6, V9\}, \{V7, V10\}, \{V8, V11\}\}.$$

This leaves:

$$P = \{V1, \ldots, V5\}.$$

This fits with the intuitive understanding of the domain. However, as yet, the tools to analyse the properties (or quality) of this decomposition have not been introduced. For that, the relationship a decomposition induces between the actions in the problem need to be analysed which is the topic of the next section.

3.3 Actions

The previous section introduced variable decompositions, which are used to split the domain into agent variables and public variables. This section discusses the possible dependencies between actions that a variable decomposition can induce in a multiagent planning problem.

The background section introduced a multiagent action classification, into internal and public actions, that is prevalent in the related literature (Section 2.3.1). This classification is fundamentally important to multiagent planning approaches and can be found, in a variety of forms, throughout the literature. In what follows, the definitions are updated for use with variable decompositions and further classifications are introduced that can be useful, both to determine the effectiveness of a particular decomposition, and in planning itself.

Action Classification For MPT $\Pi = \langle V, I, G, X, A \rangle$, variable decomposition Φ, agent i, and action $a \in A$ with $pre(a) \neq \emptyset$:
a is called an *internal action* of i iff

- $\exists v \in pre(a) : v \in \phi_i$ and

- $v \in pre(a) \rightarrow v \in \phi_i \cup P$.

a is called a *joint action* of i iff

- $\exists v \in pre(a) \ : \ v \in \phi_i$ and

- $\exists v' \in pre(a) \ : \ v \in \phi_j$ with $i \neq j$.

Finally, a is called a *public action* iff

- $v \in pre(a) \rightarrow v \in P$

The set of *internal* actions for an agent is the set of all actions that do not require, as preconditions, any variables belonging to other agents. *Public* actions are all those that only deal with environment variables. They do not belong to any agent. *Joint actions* are actions that require preconditions from multiple agents. They are generally the hardest to deal with in multiagent planning.

The action classification separates actions based on the variables in their preconditions.[1] Notice that the classification of actions does not involve their effects and that actions are classified based solely on which agent's variable sets coincide with variables in their preconditions. If an action involves a variable belonging to agent i in its preconditions, then that action either belongs solely to agent i or is a joint action between multiple agents. Some actions do not belong to any agent and are therefore public actions.

An example of an internal action is the move action in the Robots domain under the expected decomposition. This action is internal to the specific robot it is associated with because its preconditions only require variables from that agent's variable set and the public action set. This coincides with the natural interpretation that the decomposition into robots as agents would leave each move action belonging only to the relevant robot/agent. This action would not be an internal action in the definition in the literature because it changes what other agents can do. In fact, only the report action would be an internal action, making the domain appear much more complicated to solve than it actually is. The multiagent planning algorithm presented in the next chapter shows how internal actions (that are public in the traditional definition) can be used for faster planning.

[1] This classification assumes that all actions have at least one precondition.

A public action changes the environment but not the internal state of any agent. For example, in the Robots domain, adding an action that changed the location of the goal square would be a public action. The interpretation used in this thesis is that every agent can perform these public actions, though in some models it makes more sense to add an environment agent that can only perform these types of actions.

A joint action contains variables from two different agent variable sets. These are very problematic for multiagent planning because they effectively create connections directly between the agents, not in terms of the agent interactions shown in Figure 3.3, but in terms of the required inter-agent coordination to be able to perform these types of actions.

Action Sets Given an MPT $\Pi = \langle V, I, G, X, A \rangle$ and variable decomposition $\Phi = \{\phi_1, \ldots, \phi_n\}$: The action set Act_i, of agent i, is the set of all actions $a \in A$ such that a is either an internal or joint action of i. The action set *Pub* is the set of all public actions.

An action in Act_i is said to *belong* to agent i in accordance with the previous use of the term. At this point, variable decompositions induce a certain classification of the actions in the domain. There is still no part of the definition that makes a variable decomposition particularly multiagent, but, before introducing this extra layer, there are many useful properties we can prove as a direct consequence of the action definition.

Corollary 3.3.1 *For MPT* $\Pi = \langle V, I, G, X, A \rangle$ *and variable decomposition* $\Phi = \{\phi_1, \ldots, \phi_n\}$*:*

1. *If a is an* internal *action of agent i, and j is another agent* $j \neq i$*, then* $a \notin Act_j$.
2. *If a is a* joint *action of agent i, then a only appears in* Act_i *and all other* Act_j *such that* $\exists v \in pre(a) \ : \ v \in \phi_j$.
3. *For every* $a \in A$ *and* $v \in V$*:*

$$v \in pre(a) \wedge v \in \phi_i \ \rightarrow \ a \in Act_i.$$

4. *For every* $a \in A$*:*

$$a \in Act_i \ \rightarrow \ \exists v \in pre(a) : v \in \phi_i$$

5. *For* $\phi_i \in \Phi$ *and* $\phi'_j \in \Phi' = \{\phi'_1, \ldots \phi'_k\}$*:*

$$\phi_i \subseteq \phi'_j \rightarrow Act_i \subseteq Act'_j.$$

6. *For* $\phi_i \in \Phi$ *and* $\phi'_j \in \Phi' = \{\phi'_1, \ldots \phi'_k\}$*:*

$$\phi_i = \phi'_j \rightarrow Act_i = Act'_j.$$

7. *Every action appears in either some* Act_i *or in Pub. Alternatively:*

$$\Big(\bigcup_{i=1\ldots n} Act_i \Big) \cup Pub = A.$$

Proof 1-4. follow directly from the definitions.
5. Let $a \in Act_i$ then it is needed to show that $a \in Act'_j$. a can be either an internal or joint action of i but in either case $\exists v \in pre(a) : v \in \phi_i$ and this $v \in \phi_j$. Using 4 gives $a \in Act_j$.
6. follows from 5 as $\phi_i \subseteq \phi'_j$ and $\phi'_j \subseteq \phi_i$.
7. follows directly from the action classification noting that $\Phi \cup \{P\}$ is a partition of V so each $v \in V$ is either in P or some ϕ_i. □

Corollary 3.3.1.6 is particularly important as it shows that an agent's action set is independent of the composition of the other agents in the domain. Corollary 3.3.1.7 shows that every action *belongs* to some agent (or is in the public actions set).

There are some important properties of actions that cannot be taken into account without considering action effects, these are similar to the public/private actions distinction from the literature. For example, it is easier to deal with an internal action that only effects internal variables than one that effects public variables and therefore the capabilities of other agents. The current classification does not distinguish between but is extended to deal with these in Section 3.3.2.

Table 3.1 on page 59 shows the possible types of actions that can appear in a planning problem. The table separates conditions and effects that contain a single agent, multiple agents, public variables or any combination thereof. The table also distinguishes between effects that belong to agents that appear in the preconditions of an action and effects that belong to external agents (agents that don't appear in the preconditions of that action).

Of course, it would not be feasible to directly work with each separate action type that is present in the table. Furthermore, most of the action types lead to variable decompositions in which the agents cannot be thought of as separate entities as there is no part of the domain that they have autonomy over. Fortunately, the idea of agents as *entities with internal states* (Section 3.1.1) ensures that large parts of the table can be ruled out, which is formalised in the following definition.

	sAi	mAi	sAe	mAe	P	$sAi \wedge sAe$	$sAi \wedge mAe$	$mAi \wedge sAe$	$mAi \wedge mAe$	$sAi \wedge P$	$mAi \wedge P$	$sAe \wedge P$	$mAe \wedge P$	$sAi \wedge sAe \wedge P$	$sAi \wedge MAe \wedge P$	$mAi \wedge sAe \wedge P$	$mAi \wedge mAe \wedge P$
sA	I	$\times$	$I<$	$I<$	$I<$	$I<$	$I<$	$\times$	$\times$	$I<$	$\times$	$I<$	$I<$	$I<$	$I<$	$\times$	$\times$
$sA \wedge P$	$>I$	$\times$	$>I<$	$>I<$	$>I<$	$>I<$	$>I<$	$\times$	$\times$	$>I<$	$\times$	$>I<$	$>I<$	$>I<$	$>I<$	$\times$	$\times$
P	$\times$	$\times$	$>P<$	$>P<$	$>P<$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$>P<$	$>P<$	$\times$	$\times$	$\times$	$\times$
mA	J	J	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$	$J<$
$mA \wedge P$	$>J$	$>J$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$	$>J<$

Table 3.1: Table showing the possible action types in a planning problem. The left column represents the sets which contain variables from the preconditions of the action, the top row shows which sets the effect conditions contain variables from. s means single, m means multiple, i means internal and e means external. So mAe represents that the effects contain variables from multiple agent sets that are not also involved in the preconditions. The colours represent the following: green = relevant to a single agent; blue = relevant to all agents; red = relevant to all agents and requiring coordination of internal agent states. Orange coloured are actions that manipulate other agent's variable sets. $>$ denotes an influenced action and $<$ denotes an influencing action. This classification is explained in Section 3.3.2

Agent Property For MPT $\Pi = \langle V, I, G, X, A \rangle$ and variable decomposition Φ, a set $\phi_i \in \Phi$ has the *agent property* if for all actions $a \in A$ and variables $v \in V$:

$$v \in \phi_i \wedge v \in eff(a) \rightarrow a \in Act_i.$$

Due to Corollary 3.3.1.4 it is possible to rewrite the agent property condition as:

$$v \in \phi_i \wedge v \in eff(a) \rightarrow \exists v' \in pre(a) : v' \in \phi_i$$

This definition ensures that variables in an agent's variable set ϕ_i do not appear in the effects of actions that do not belong to that agent. In particular, it ensures that the variable decomposition has the form depicted on the right hand side of Figure 3.3. In other words, under the agent property, an agent's variables can be considered as *internal* as only actions belonging to that agent can change them. Corollary 3.3.1.6 ensures that whether or not an agent set has the *agent property* is independent of the other agent sets in the domain. This is an important result as it means that agent sets can be found independently, a fact used in the decomposition algorithm presented in Chapter 4. If there are at least two agents, and all agent sets in a variable decomposition have the agent property, then it is called an agent variable decomposition.

Agent Variable Decomposition A variable decomposition Φ of MPT $\Pi = \langle V, I, G, X, A \rangle$ is called an *Agent Variable Decomposition* (AVD), or *agent decomposition* for short, if $|\Phi| > 1$ and every agent set $\phi_i \in \Phi$ has the *agent property*.

An AVD is the first step towards defining a multiagent MPT, it defines a partitioning of the variable set such that each agent set represents the internal state of an agent and has the agent property, and there is a leftover public variable set that agents can interact with or manipulate. However, as the definition stands, any given planning problem can have, and in most cases will have, multiple associated AVDs. For example, the following corollary shows how agents can be combined to artificially create new AVDs.

Corollary 3.3.2 *For MPT $\Pi = \langle V, I, G, X, A \rangle$, if ϕ_i has the agent property and ϕ_j has the agent property then $\phi_k = \phi_i \cup \phi_j$ has the agent property.*

Proof This follows from showing that $Act_k = Act_i \cup Act_j$ which in turn follows directly from 3.3.1.3. □

This corollary shows that even in well-behaving AVDs, where all actions are *internal*, there will likely be multiple competing AVDs. The fact that agents can be arbitrarily

combined suggests that the most fundamental AVDs are those that contain the most agents. It follows from the independence of the agent property (Corollary 3.3.1.6) that if $\Phi = \{\phi_1, \ldots, \phi_n\}$ is an AVD then $\Phi' = \{\phi_1, \ldots, \phi_{n-1}\}$ is also an AVD as long as $n > 1$. It is natural to think of the former as the better agent decomposition because this one assigns more variables to agents. This leads to the following definition:

Definition A *maximal* AVD for an MPT Π is any agent decomposition Φ with public variable set P that satisfies the following:

- For all other AVDs Φ' with public variable set P', $|P| \leq |P'|$.
- For all other agent decompositions Φ' with public variable set P' such that $|P| = |P'|$, $|\Phi| > |\Phi'|$.

This definition ensures that as many variables as possible are contained in agent's internal states. In other words, the number of variables left public is minimised. Furthermore, decompositions that minimise the number of public variables must maximise the number of agents. It is important to minimise the number of public variables, as this reduces the possible size of the coordination problem between the agents and also ensures that no potential agent is left out of the decomposition.

An important class of multiagent planning problems are those without joint actions. Joint actions are problematic because it is usually possible to split an agent variable set apart into multiple separate sets that still have the agent property but contain a lot of joint actions between them. Decompositions with no joint actions ensure that all coordination is via the public variables and are much easier to deal with.

Definition A maximal *separated* AVD for an MPT Π is any agent decomposition Φ with public variable set P that contains no joint actions and is a *maximal* AVD amongst all those with no joint actions. An MPT Π, along with a maximal separated AVD Φ is an MMPT (a multiagent MPT), and is represented by $\Pi = \langle V, \Phi, I, G, X, A \rangle$.

Separated AVD's have much stronger properties than their non-separated counterparts because the agent property is linked to the action set of an agent Act_i which includes any joint actions that an agent might have. Once joint actions are removed from the equation, the set Act_i only contains internal actions of an agent. This means that, in a separated AVD, if an action's preconditions contain a variable belonging to an agent, then all other variables in its preconditions belong to that agent or the public set and similarly, all variables in its effects must belong to either that agent or the public variable set. This is summed up in the following corollary.

Corollary 3.3.3 *For MPT* Π *and maximal separated AVD* Φ*. For any action* $a \in A$*:*

$$\exists v \in pre(a) : v \in \phi_i \rightarrow \forall v' \in \big(pre(a) \cup eff(a)\big) : v' \in \phi_i \cup P$$

Note that working with separated AVDs technically does not reduce the number of potential problems with a decomposition as it is possible to convert AVDs into separated-AVDs by rewriting the planning problem. Any joint actions can be rewritten as internal actions for each agent, and sufficient new public variables can be introduced to ensure that the separated parts of the joint actions always occur together. However, this result is only of technical relevance as the actual conversion is not practical as it exponentially increases the size of the domain.

This section argued that an agent's variable set should represent an internal state of that agent, and AVDs were defined based on this assumption. A *maximal-separated* AVD was defined as an AVD that minimises the number of public variables while maximising the number of agents amongst all AVDs that contain no joint actions. A maximal-separated AVD, along with its associated MPT, is called an MMPT. This will become the fundamental definition for the multiagent planning domains presented in this thesis. The next section shows how these definitions relate to the example Robots domain.

3.3.1 Example: Toy Robot Domain - Agent Variable Decomposition

Recall that the last visit to this example domain suggested the variable decomposition

$$\Phi = \{\{V6, V9\}, \{V7, V10\}, \{V8, V11\}\}.$$

This decomposition is indeed an agent variable decomposition. In fact it is the unique *maximal-separated* agent variable decomposition for the domain.

There are only two operators in the domain, so it is easy to confirm that this is an AVD:

- `move` operator: This operator takes as conditions variables from the public set and variables from one agent's set. It affects variables from the public set along with variables from the same agent's set. Therefore this operator does not break the agent property.

- `report` operator: This operator takes as conditions variables from one agent's set. It affects only variables from the same agent's set. Therefore this operator does not break the agent property.

There are other possible agent variable decompositions of the domain, the most intuitive being:

$$\Phi = \{\{V6\},\{V7\},\{V8\}\}.$$

This decomposition moves all the 'reported' variables into the public actions set and is also valid. This decomposition has a larger public variable set so is not maximal. Other valid decompositions would be:

$$\Phi = \{\{V7\},\{V8\}\}$$

or

$$\Phi = \{\{V6,V7\},\{V8\}\}.$$

On the other hand, the decomposition

$$\Phi = \{\{V1\},\{V2\},\{V3\},\{V4\},\{V5\}\}$$

is not an AVD. This is because, for example, the action `(move a x y)` breaks the agent property by having $V2$ in its conditions and $V1$ in its effects.

The action classification given so far is enough to define maximal agent variable decompositions. However, Table 3.1 showed further distinctions between actions that are important in determining the difficulty of a multiagent planning problem. These further distinctions are between *influenced* and *influencing* actions, which show how an agent's actions interact with the rest of the domain.

3.3.2 Influenced and Influencing Actions

The AVD definition drastically reduces the number of possible action types from those shown in Table 3.1. The agent property ensures that it is not possible to have actions with external agents in their effects. This greatly simplifies the types of possible actions leading to the reduced table shown in Table 3.2 on page 64, which is even further reduced in separated-decompositions.

Without the possibility to influence another agent's internal state (except through joint actions shared with them), agents can only interact through the public variables in the domain. The following definitions separate the actions by how they interact through the public action set, which allows for a useful estimate of the complexity of the coordination problem of an AVD.

Influenced An action $a \in A$ is *influenced* if:

$$\exists v \in pre(a) : v \in P$$

	A_i	$A_i \wedge P$	P	mA_i	$mAi \wedge P$
A_i	I	$I <$	$I <$	$\times$	$\times$
$A_i \wedge P$	$> I$	$> I <$	$> I <$	$\times$	$\times$
P	$\times$	$\times$	$> P <$	$\times$	$\times$
mA	J	$J <$	$J <$	J	$J <$
$mA \wedge P$	$> J$	$> J$	$> J <$	$> J <$	$> J <$

Table 3.2: Table showing the possible types of actions in a MMPT. The left-side column shows the agent sets present in the preconditions of the action and the top row shows the agent sets present in its effects. A $>$ symbol denotes an action as being *influenced*, a $<$ as *influencing*. I represents internal and partially internal actions, P represents public actions and J represents joint actions. A_i means the single agent A_i is involved in the action while mA means that multiple agents are. A $\times$ symbol shows where actions are impossible.

Influencing An action $a \in A$ is *influencing* if:

$$\exists v \in \mathit{eff}(a) : v \in P$$

An action is *influenced* if some of its conditions can potentially be changed by other agents in the domain. An action is *influencing* if it can potentially effect which actions are possible by other agents in the domain. All public actions are, by definition, both influenced and influencing.

The agent property ensures that agents can only interact through joint actions or through changes to the environment (public variables) while the influenced/ing categorisation describes wether an action can indirectly effect others or can be indirectly effected by others. The hardest actions to deal with are both influenced and influencing. On the other hand, some actions are neither influenced or influencing, these actions are the easiest to deal with as they do not contribute to the coordination problem between the agents.

3.3.3 Example: Toy Robot Domain - Actions

The example domain is too small to include each possible type of action. The following shows how it could be modified to include each possible action type under the maximal agent variable decomposition

$$\{\{V6, V9\}\{V7, V10\}\{V8, V11\}\}.$$

- (I) Each `report` action is internal and neither influenced or influencing. For example, the action `report(a, x)` requires a particular value for V6 and changes only the value of V9. Because both V6 and V9 are in the same variable set then this is an internal action.

- $(I <)$ If the `move` action for robot `a` is modified so that this robot could move into grid squares that contain other agents, but they could still not move into grid squares that contain it, then this new move action would be internal and influencing but not influenced.

- $(> I)$ If the `move` action for robot `a` is modified so that other robots could move into the location that it occupies, but it still could not move in to locations that contain other agents, then the new `move` action would be internal and influenced but not influencing.

- $(> I <)$ Each `move` action is internal and both influenced and influencing. For example, the action `move(a, x, y)` requires a specific value for V6 and V2 and changes V6, V2 and V1. V2 is in P but V6 is in A_1 and the action effects the value of elements of A_1 and P.

- $(> P <)$ An action that changes whether (or not) a particular grid square is free would be a public action. For example, consider an action `makefree(x)` that has no preconditions and sets the value of V1 to $\perp$. Public actions are necessarily influenced and influencing.

- (J) A `swap` action by which two agents could exchange places would be a joint action. This action would be neither influenced or influencing as it would not change or depend on the value of any of the public variables in the domain. The grid squares each agent started on would remain not free.

- $(J <)$ A `create-blockage` action by which two robots in the same space could mark a location as not free (without requiring it to be free in the first place) would be joint and influencing but not influenced.

- $(> J)$ If the `swap` action was changed so that it could only be done when a particular location is free then it would be joint and influenced but not influencing.

- $(> J <)$ A `joint-move` action by which two robots moved simultaneously to different locations would be both influenced and influencing.

Even though there are still a lot of different action types, in separated AVDs the most important distinction is only between internal and public actions. An agent's actions are only important to the coordination problem in the way in which they interact with the public variable set, which is another reason that the public variable set should be minimised in *maximal* agent variable decompositions. The next section collects together the results of this chapter to define MMPTs (multiagent multi-valued planing tasks).

3.4 Summary

This chapter defined MMPTs, a formalism for multiagent planning based on the MPT representation. In MMPTs, agents are defined by a collection of variables that represent their internal state, in that, these variables are not modifiable by the actions of any other agent in the domain. The most important type of MMPTs were maximal-separated MMPTs that assign the maximum number of variables to agents as possible and contain no joint actions. These will become the focus of most of the methods presented in the rest of the thesis.

Chapter 4

Algorithms for Multiagent Classical Planning

This chapter presents the main contributions of the thesis; an automated decomposition algorithm, and a heuristic multiagent plan search algorithm. The background chapter showed that even though there are several, distinct, multiagent planning approaches with promising results, most planning domains are written, represented and solved in a single-agent manner with one large collection of actions. The first part of this chapter presents an automated agent decomposition algorithm that can find separated AVDs from standard single-agent classical planning problems written in PDDL. This means that multiagent planning techniques can be used on existing single-agent planning domains that have an inherent multiagent structure, without requiring a human expert with full domain knowledge to manually separate the agents.

With a decomposition algorithm in place, the next task is to try and exploit the multiagent structure afforded by the AVD representation to improve planning times. The second part of this chapter introduces a heuristic multiagent planning algorithm that takes a separated AVD as input, and outputs a solution to the planning problem, if one exists. The algorithms presented in this chapter are empirically evaluated in Chapter 5 and the multiagent planning algorithm is shown to greatly outperform state-of-the-art planners on decomposable domains.

4.1 Agent Decomposition

The decomposition algorithm is based on analysing the causal graph of the planning problem. Causal graphs, introduced in Section 2.1.4.2, are an important structure in

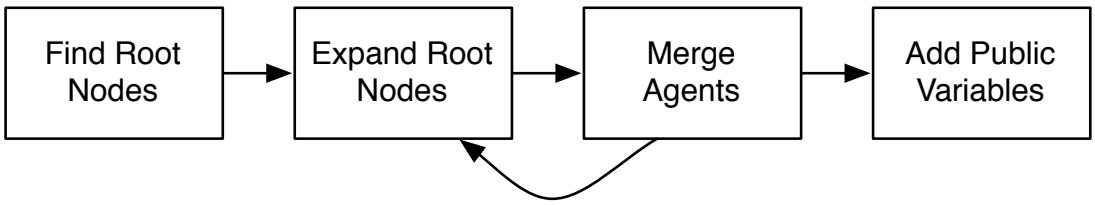

Figure 4.1: Overview of the agent decomposition algorithm

recent single-agent planning approaches and encode dependencies between the variables, induced by the actions in a domain. These dependencies can be analysed to find sets of variables that can represent internal states of agents.

The input for the overarching process is a classical planning problem specified in PDDL. This is then parsed by the Fast Downward planning system to create an MPT representation along with the causal graphs (modified as detailed in the next section), which are then passed as input for the decomposition algorithm. The output of the decomposition process is a separated AVD (Section 3.3) if one exists, a representation of the problem with variables and actions decomposed into separate agent sets.

The decomposition algorithm is split into four main parts, shown schematically in Figure 4.1. The first part analysis the modified causal graph to find root nodes. Root nodes in the modified causal graph, if they exist, have the property that they cannot be in the effects of actions that depend on the value of other variables. Therefore, a singleton set containing only a root node of the causal graph has the agent property, and can be used as a starting point for creating a separated AVD.

The second part of the algorithm extends the root nodes to their neighbours, maximising the number of internal variables for each agent. This process follows the edges in the causal graph, ensuring that certain properties hold, in order to preserve the agent property. However, this process does not necessarily lead to a separated AVD as it can create an AVD with joint actions. Therefore, the third part merges agent sets in such a way as to remove joint actions. The second and third parts of the algorithm are repeated until either a separated AVD, or a single set is left. In the former case, with leftover variables becoming the public variables in the domain, the algorithm returns a separated AVD, while in the latter case it returns that none can be found.

4.1.1 Causal Graphs

As was discussed in Section 2.1.4.2, there already exist methods for calculating the dependencies between variables induced by the actions in an MPT. These are called

causal graphs and are used in many different (single-agent) planning approaches in order to help navigate through a problem's search space.

Definition Let $\Pi = \langle V, I, G, X, A \rangle$ be an MPT. The causal graph of Π, written as $CG(\Pi)$, is the directed graph with vertex set V containing an arc (v, v') iff $v \neq v'$ and one of the following holds:

- **Transition condition** There is an action (or axiom) that can affect the value of v' which requires a value for v in its precondition (or condition).
- **Co-occurring effects** The set of affected variables in the effect list of some operator includes both v and v'.

In other words, based on the transition condition, if there exists an edge between v and v' in the causal graph then

$$\exists a \in A : v' \in \mathit{eff}(a) \wedge v \in \mathit{pre}(a)$$

However, the causal graph definition needs to be modified slightly for use with the decomposition algorithm. The transition condition is still an important concept, but two-way edges caused by the same action must be ignored. Also, co-occurring effects are not included. The modification is that if a single action induces an edge (v, v') and also an edge (v', v), then both edges are ignored. It is still possible for edges (v, v') and (v', v) to exist in the causal graph, but they must have come from separate actions. In what follows, all references to causal graphs are to the modified causal graph that does not include such edges.

Definition Let $\Pi = \langle V, I, G, X, A \rangle$ be an MPT. The *modified* causal graph of Π, written as $CG(\Pi)$, is the directed graph with vertex set V containing an arc (v, v') iff $v \neq v'$ and the following holds:

- **Modified Transition condition** There is an action (or axiom) that can affect the value of v' which requires a value for v in its precondition (or condition) and that action does not affect the value of v while also having v' in its precondition.

The modified causal graph of a problem shows the dependencies between variables based on the actions in the problem. Variables are linked when they appear with one in the effects and the other in the preconditions of an action. This is clearly linked to the agent property condition which required for agent set ϕ_i and any action $a \in A$ that:

$$v \in \phi_i \wedge v \in \mathit{eff}(a) \rightarrow \exists v' \in \mathit{pre}(a) : v' \in \phi_i$$

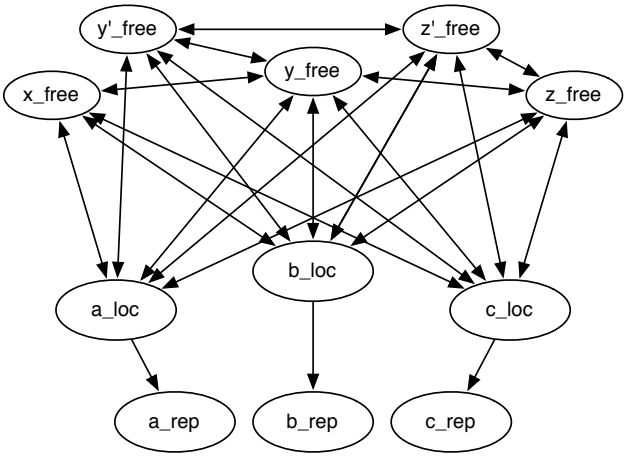

Figure 4.2: The causal graph for the problem shown in Figure 3.1.

However, while there is certainly a relationship between causal graphs and the agent property, it is not immediately obvious what properties this relationship has. In order to provide a more intuitive understanding, the next section describes the causal graph for the Robots example problem and discusses how this relates to the AVDs of the domain. This is followed by a more rigorous analysis of the relationship between the modified causal graph and agent decompositions.

Example: Toy Robot Domain

Figure 4.2 shows the (standard) causal graph for the robot domain that has been used as an example throughout this thesis. The diagram shows that the location variables are strongly connected. The variables that represent whether or not a robot has reported, on the other hand, are only linked to the same robot's location. It is easy to see from the figure that the problem has some underlying structure that makes it amenable to a multiagent approach. In this case the structure is more important in the absence of connections than in the connections themselves.

On the other hand, the modified causal graph displays an interesting structure. The modified causal graph is shown in Figure 4.3, (nodes that do not have any incoming or outgoing edges have been omitted). From this version of the graph, it is very easy to see how the agents are split up and the disconnected subgraphs can literally be used to create the agent variable sets.

It is important not to generalise too much from the casual graph of the robot domain. Most causal graphs are not symmetric and the patterns that occur here are due to the simplicity of the problem instance. It is not even possible to provide a visual representa-

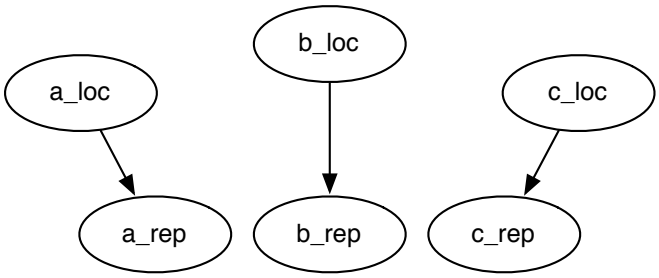

Figure 4.3: The modified causal graph for the robot domain.

tion of the causal graph for any non-trivial domain. Standard planning problems contain hundreds, or thousands, of variables and even very small problem instances will have too many connections to be represented visually. A further understanding of causal graphs can only be developed by looking at their formal properties.

4.1.2 Causal Graphs and Agents

This section gives some immediate properties of the relationship between causal graphs and AVDs which will be utilised in the decomposition algorithm. The first result shows that if a link appears from v to v' in the causal graph then v and v' cannot belong to separate agents.

Proposition 4.1.1 *For AVD Π^* with separated agent variable decomposition ϕ, if $(v, v') \in CG(\Pi^*)$ with $v \in \phi_i$ then $v' \in \phi_i \cup P$.*

Proof Assume, by way of contradiction, that $v' \notin \phi_i \cup P$, then $v' \in \phi_j$ for some $j \neq i$. Let a be an action that induces the edge $(v, v') \in CG(\Pi)$. Then $v' \in \mathit{eff}(a) \wedge v \in \mathit{pre}(a)$. As Π^* is separated, a must be an internal action of ϕ_i. But then v' is in its effects and belongs to another agent ϕ_j which breaks the agent property and we have a contradiction. □

This means that any connected parts of the causal graph must either belong to the same agent, or at least one of them must belong to the public variable set. This is a strong result towards forming an algorithm for finding decompositions. However, it is useless without a starting point that assigns some variables to an agent. The next result shows that root nodes in the standard causal graph will always have the agent property, and therefore that they can act as starting points for generating a decomposition, assuming they exist.

Proposition 4.1.2 *If* $\{v_1, \ldots, v_n\}$ *are all root nodes of a (standard) causal graph CG for MPT* Π*, then* $\Phi = \{\{v_1\}, \ldots, \{v_n\}\}$ *is an agent variable decomposition of* Π*.*

Proof Assume, by way of contradiction, that a is an action that would break the agent property. Choose $\phi_i \in \Phi$ and $v_i \in V$ such that $v_i \in \phi_i \wedge v_i \in \mathit{eff}(a) \wedge a \notin \mathit{Act}_i$. If $\mathit{pre}(a) \neq \emptyset$ then choose $v' \in \mathit{pre}(a)$. In this case $(v', v) \in CG$ (as $v' \in \mathit{pre}(a)$ and $v \in \mathit{eff}(a)$), so v cannot be a root node of the causal graph and we have a contradiction. On the other hand, if $\mathit{pre}(a) = \emptyset$ then we classify a by using $\mathit{eff}(a)$ in place of $\mathit{pre}(a)$. Because $v_i \in \mathit{eff}(a)$ the action must belong to Act_i, which contradicts $a \notin \mathit{Act}_i$. □

The proposition makes intuitive sense. Each root node of the causal graph, by virtue of having no incoming edges, is a variable that is not affected by any actions in the domain. This means that the variable cannot possibly break the agent property, as that is defined based on action effects. Variables such as this can naturally become part of the internal state of some element of the environment.

This result means that taking all root nodes of the standard causal graph will give us an agent variable decomposition for a problem. However, this is unlikely to be close to the maximal agent variable decomposition. Even worse, there are many problems for which the standard causal graph contains no root nodes. For example, the example problem used throughout the previous chapter, with agent variable decomposition $\{\{V_6\}, \{V_7\}, \{V_8\}\}$, does not contain any root nodes in its standard causal graph.

This means that the previous result cannot be used to provide a method for finding variables to become part of an agent decomposition in the general case. On the other hand, the modified causal graph may contain root nodes where the standard causal graph does not, and the following proposition shows that these root nodes can be used to form an agent variable decomposition, as long as there are no joint actions. This result is the reason that the decomposition process focusses on separated AVD's, as the agent property is only preserved when working with modified causal graphs under the assumption that there are no joint actions. In the following proposition, by non-singleton it is meant that a node has at least one incoming or outgoing edge.

Proposition 4.1.3 *If* $\{v_1, \ldots, v_n\}$ *are all non-singleton root nodes of the* modified *causal graph CG for MPT* Π*, then* $\Phi = \{\{v_1\}, \ldots, \{v_n\}\}$ *is an agent variable decomposition of* Π *as long as there are no joint actions.*

Proof As in the previous proof assume, by way of contradiction, that a is an action that would break the agent property. Choose $\phi_i \in \Phi$ and $v_i \in V$ such that $v_i \in \phi_i \wedge v_i \in$

$\mathit{eff}(a) \wedge a \notin \mathit{Act}_i$. The interesting case is when a two-way edge was removed. If the action a was not part of removed edges, then we have the same proof as the previous proposition. On the other hand, if v' belongs to the public action set, then we are fine, but if it belongs to another agent set, then we must have a joint action which is a contradiction. □

This result may not seem powerful at first, as the requirement that there are no joint actions induced by the variable decomposition is significantly limiting. However, if joint actions are formed, then all is not lost. It is possible to merge the agents responsible for creating the joint actions at a later stage in such a way as to regain the agent property. However, first it is shown how agent sets can be extended whilst still preserving the agent property.

Recall that maximal agent decompositions require the agent sets to be as large as possible. The current results only allow for the creation of singleton agent sets. The following definition and proposition shows how successors in the causal graph can be used to increase the size of agent variable sets while maintaining the agent property.

Definition An *agent successor* of an agent set ϕ is any variable that is a successor of some element of ϕ and that has no predecessors not in ϕ.

Let $\mathit{succ}(\phi)$ represent all the agent successors of ϕ.

Proposition 4.1.4 *If* $\{\phi_1, \ldots, \phi_n\}$ *is an agent variable decomposition, then*

$$\{\phi_1 \cup \mathit{succ}(\phi_1), \phi_2, \ldots, \phi_n\}$$

is an agent variable decomposition.

Proof This is a direct consequence of the modified Causal Graph definition. □

This result means that it is possible to extend an agent variable decomposition, as long as there remain agent successors in the causal graph. The process can be repeated until no such possible extensions remain. Starting with all the root nodes of the modified causal graph and then repeatedly extending to agent successors will produce an agent variable decomposition (ignoring joint actions).

As has been discussed, joint actions are problematic as they can break the agent property. Unfortunately, they appear frequently because the set of root nodes of the modified causal graph tends to overestimate the possible agents in the domain. However,

it was shown in Corollary 3.3.2 that two agent sets can be combined without affecting the agent property. This means that when two agents that share a joint action are combined into one, then (as long as there are no further joint actions) the resulting agent must have the agent property.

Proposition 4.1.5 *If each ϕ_i in $\Phi = \{\phi_1, \ldots, \phi_n\}$ has the agent property when ignoring joint actions then ϕ'_j which is formed from the union of members of Φ that share joint actions has the agent property (and no joint actions).*

Proof The first part of this was shown in Corollary 3.3.2. The second part (that there are no joint actions) is clear from the fact that any agent that may have shared a joint action with ϕ'_j is now joined with ϕ'_j.

By combining agents that share joint actions in this way, all joint actions are removed from the domain, leaving a separated agent decomposition. With these results it is possible to create an algorithm for finding separated agent decompositions. It is not known at this time if the returned decompositions are necessarily maximal, although the empirical results presented in the following chapter suggest that they might be.

4.1.3 Agent Decomposition Algorithm

This section introduces an agent decomposition algorithm that builds on the previous results and has the overall structure shown in Figure 4.1. Algorithm 1 shows the pseudocode for this process. The agent set is initialised to the empty set (line 1) and then all root nodes of the modified causal graph are added as agent sets (line 2). The details of this process are given in the Function FindRootNodes on page 75. Then, the ExtendAgentSets (page 76) and MergeAgents (page 77) methods are called repeatedly until there is no longer any change in the agent set. In practice they are usually only called two or three times.

Root Nodes

The method for finding root nodes in the modified causal graph is shown in pseudocode Function FindRootNodes. Starting with the empty set as input, each variable is checked in turn to determine if it is a root node of the modified causal graph. This is achieved (line 2) by ensuring that there are no predecessors in the causal graph and that there is at least one successor. The function nodes(CG) returns the nodes of the causal graph

Algorithm 1: Agent Decomposition Algorithm

Input : Modified Causal Graph CG, MPT $\Pi = \langle V, I, G, X, A \rangle$

Output : $\langle V, \Phi, I, G, X, A \rangle$ // A separated-AVD of Π

```
1 Φ ←FindRootNodes(CG)                          // Defined on p.75
2 repeat
3 |   Φ' ← copy(Φ)
4 |   Φ ← ExtendAgentSets(Φ, CG)                 // Defined on p.76
5 |   Φ ← MergeAgents(Φ, CG)                     // Defined on p.77
6 until Φ' ≡ Φ
7 return Φ
```

CG, which will be equal to V from the original MPT. The functions $pre(v, CG)$ and $succ(v, CG)$ return respectively the set of predecessors or successors of a node v in the modified causal graph CG. The successor is required to make sure that the node is not a singleton node of the reduced causal graph. If all the checks pass, then the node in question is a root node of the modified causal graph and is added to the current agent set.

Function FindRootNodes(CG)

```
1 Φ ← ∅                                         // Initialise agent set
2 foreach v ∈ nodes(CG) do
3 |   if pre(v, CG) = ∅ ∧ succ(v, CG) ≠ ∅ then
4 |   |   Φ ← Φ ∪ {{v}}                        // i.e. {v} is a new agent set.
5 return Φ
```

The output of this function is a variable decomposition composed of only singleton variables. At this stage of the algorithm the variable decomposition is only guaranteed to be an agent variable decomposition if there are no joint actions induced by it (see Proposition 4.1.3). The next function extends the singleton agent sets to form larger agents.

Extending Variable Sets

In this part, the agent sets are extended recursively as shown in the Functions ExtendAgentSets and Extend on page 76. For each variable that belongs to an agent set, its

Function ExtendAgentSets(Φ, CG)

```
1 foreach ϕ ∈ Φ : v ∈ ϕ do
2 |   ϕ ← Extend(v, ϕ, CG)
3 return Φ
```

Function Extend(v, ϕ, CG)

```
1 foreach s ∈ succ(v, CG) do
2 |   if ∀(p, s) ∈ CG : p ∈ ϕ then
3 |   |   ϕ ← ϕ ∪ s
4 |   |   ϕ ← Extend(s, ϕ, CG)
5 return ϕ
```

successors are checked to see if they satisfy the agent successor condition and can be added to the variable set. If a variable gets added, then it, in turn, will also be expanded. It should be noted at this point that the resultant agent sets are guaranteed to be a partitioning of V (with the leftover variables as member of the public variable set). This is because it is not possible for a variable to be the successor of two different agent sets starting from two different root nodes by definition.

Once this process is completed there will be the same number of agent sets, but they may each haven grown in size. While the size of the agent sets may have increased, there may still be joint actions and so the current decomposition set may still not form a valid agent variable decomposition. The next part of the algorithm merges agents to ensure that the agent variable decomposition is separated and therefore that the agent property holds (Proposition 4.1.5).

Merging Agents

The MergeAgents function is shown on page 77. This is a relatively simple process that involves iterating over all the actions in the domain. The action's variables are checked to see if they belong to any of the agent sets. If they contain variables that belong to multiple agents, then those agents' variable sets are merged together. This process removes all joint actions from the decomposition, and leaves either a separated agent variable decomposition of the domain or a decomposition containing only one agent. In the latter case, the algorithm returns that it cannot find a separated AVD.

The results from the previous section showed that the algorithm will return either a

Function MergeAgents($\Phi, \Pi = \{V, I, G, X, A\}$)

```
1 foreach a ∈ A do
2     φ_a ← {φ_i ∈ Φ | a ∈ Act_i}
3     if |φ_a| > 1                          // i.e. a is a joint action
4     then
5         Φ ← Φ \ φ_a
6         Φ ← Φ ∪ {⋃φ_a}
7 return Φ
```

separated AVD or, when either no root nodes can be found, or all agent sets become merged, it returns that none can be found. It is not known at this time whether or not the algorithm is complete. The empirical results show that the algorithm returns a separated AVD in all the expected cases from the IPC domains and it is believed that future work will show that this algorithm (perhaps with some very minor modifications) is indeed complete. The next section discusses how this algorithm works on the Robots example problem.

4.1.4 Decomposition Algorithm for the Robots Example

In the Robots example domain from the previous chapters, robot's had to navigate a gridworld and report to a goal location. The causal graph for the example problem was shown in Figure 4.2 and has the property that it contains no root nodes. The first part of the decomposition algorithm finds the root nodes in the modified causal graph, which effectively finds the root nodes of the modifid graph shown in Figure 4.3. This is a much simpler graph, which has root nodes that correspond to the robot's locations. Using the variable names given in Figure 3.4, the output of the first part of the algorithm is

$$\{\{V6\},\{V7\},\{V8\}\}.$$

The second part of the decomposition extends the variable sets to agent successors. From Figure 4.3 it is easy to see that each of the variables found so far has a single agent successor. In more complicated problems, the extension process may end up adding many variables, many edges away from the starting variable for the agent. Returning the the Robots example, the output of this part of the algorithm is

$$\{\{V6,V9\},\{V7,V10\},\{V8,V11\}\}.$$

There are no joint actions created by this decomposition so the merging part of the decomposition has no effect. The final output is therefore the decomposition

$$\{\{V6, V9\}, \{V7, V10\}, \{V8, V11\}\}$$

with

$$\{V1, V2, V3, V4, V5\}$$

forming the public variable set. In a more complicated problem, there may have been joint actions that caused certain variable sets to be merged. If this happened, then the algorithm would repeat the extension and merging phase once more.

4.1.5 Summary

This section presented an algorithm for computing separated AVDs for planning problems. The input for this algorithm can be any standard planning problem formatted in PDDL that can be converted into an MPT representation, which covers all the classical planning domains used in the International Planning Competition's main track. The algorithm is analysed empirically in Chapter 5.

4.2 ADP - A Multiagent Heuristic Planning Algorithm

The previous work has provided a formalism for multiagent planning and shown how to compute separated AVDs from standard single-agent planning problems. The next step is to find methods for solving these separated AVDs. This section introduces a cooperative, centralised, heuristic planning algorithm that will efficiently find a plan for any separated AVD. The algorithm described in this section assumes that the agents in the domain are working towards a shared goal and that they do not care how many of 'their actions' are in the final plan. The only important thing is that the final plan results in the shared goal being met.

The algorithm presented in this section, ADP (which stands for Agent Decomposition Planner) is based on the famous no delete list heuristic, which, as mentioned in the background chapter, has proved very successful in single-agent planning approaches (Bonet and Geffner, 1999; Hoffmann and Nebel, 2001). In fact, if the algorithm in this section is given a single-agent MPT as input, then it performs similarly to the FF algorithm, albeit with some unnecessary calculations. In cases where there exists a proper decomposition, then, in the vast majority of cases, the algorithm presented in this section greatly outperforms its single-agent counterpart in terms of planning time. The empirical evaluation of this is presented in Chapter 5.

ADP consists of a heuristic calculation method for greedy best-first search. As with all heuristic-based searches, the effectiveness of ADP is dependent on the quality of the heuristic, and the time that it takes to compute it. The heuristic value of a state is based on the global progress of all agents to the goal (h_G), which is only updated at certain *coordination points*, and the local progress of a particular agent towards a set of subgoals that have been chosen as important to find next (h_L), the calculation of which only requires use of that single agent's subproblem. The general idea behind ADP is to reduce heuristic calculation time by only working with agent's individual problems. Heuristic quality is maintained, or even improved, by keeping track of the global progress whilst also including an agent's progress towards its subgoals which may help solve parts of the problem that require agent interaction.

In multiagent planning, the limiting factor is usually solving the coordination between the agents. This is why most multiagent planning work focusses on cases where there is little, or reduced, coordination. However, in ADP, by converting the planning problem into a collection of individual problems, the search algorithm never has to deal with the coordination problem directly. In fact, ADP never has to search for

Algorithm 2: ADP Search Algorithm

Input : Separated–AVD $\Pi = \langle V, \Phi, I, G, X, A \rangle$

Output : π `// Either a plan for` G `or [] if no plan exists.`

```
1  I.⟨h_G, φ_i, G_i, h_L⟩ ← CalculateHeuristic(I, Π)        // See p.82.
2  open ← [I]
3  closed ← []
4  while open ≠ [] do
5      S ← open.pop        // S will have best found heursitic value.
6      if G reached in S then
7          π ← Calculate plan by chaining back through S.pre to I
8          return π                                  // A plan has been found.
9      foreach Successor S' of S do
10         if S' ∉ closed then
11             S'.pre ← S
12             S'.⟨h_G, φ_i, G_i, h_L⟩ ← CalculateHeuristic(S', Π)    // See p.82.
13             open.insert(S', S'.h_G + S'.h_L)   // Insert based on heuristic.
14             if S'.h_G + S'.h_L is the best heuristic so far then
15                 open.insert(S, S.h_G + S.h_L)              // Reinsert S.
16                 break   // Do not continue to expand S (move to S').
17     if All successors of S generated then
18         closed.add(S)
19 return π
```

more than one agent at a time, even during the calculation of subgoals at coordination points. This is why ADP is shown to be effective even in planning problems which require a lot of coordination between agents. The details of the coordination point calculation are given in Section 4.2.4.

4.2.1 ADP - Algorithm

ADP is a forward state-space search algorithm that utilises the common "no delete lists" heuristic. The overall search process used by ADP is greedy-best-first search, which always expands the node with the lowest (best) stored heuristic values and maintains a *closed* list of states to avoid becoming caught in loops. Pseudocode for this process is

shown in Algorithm 2. The heuristic value of a state

$$h = h_G + h_L$$

is calculated as a summation of the global heuristic value (h_G), which has been carried over from the last coordination point, and the local heuristic value of the current agent towards its subgoals (h_L).

The differences between Algorithm 2 and a standard greedy best-first search implementation are only in the extra values that must be maintained with a state.

- h_G is the global heuristic value. This value will only change at coordination points. Otherwise, it is carried over from the previous state.
- ϕ_i is the currently chosen agent. This is also carried over from state to successor state and is only changed at coordination points.
- G_i is the goal set of the currently chosen agent. It behaves like the previous two items. G_i may be a subset of G (the goal set of the problem) but may also include subgoals that do not belong in G. These additional goals are calculated at coordination points as required for other agent's to be able to progress the solution.
- h_L is the local heuristic value of ϕ_i towards G_i. The value of this will change from state to state as the currently selected agent updates its individual heuristic value for searching towards its subgoals.

Note that for the comparison in line 11 of a state with those in the closed list, then only the elements of the actual state are compared, not the values h_G, ϕ_i, G_i, and h_L.

Ultimately, the multiagent decomposition is only used to calculate a heuristic value for greedy-best-first search and no states are pruned based on local dead ends. This means that, like the overall search algorithm, ADP is both sound and complete. However, due to its heuristic nature, in the worst case the entire search space is covered. The effectiveness of ADP is dependent on the route that the heuristic values create through the search space along with the time it takes to calculate the heuristic values and coordination points.

The next section introduces the heuristic calculation method. If the current state is not a coordination point, then all that is needed to be done is calculate a new local heuristic value for the current agent. At coordination points, a new global heuristic value (h_G) is calculated along with a new agent (ϕ_i) and a set of subgoals (G_i). These are then used to calculate the new agent's local heuristic value.

Function CalculateHeuristic(S, Π)

```
1  if S ≡ I then
2  |  h_L ← ∞                                  // Only happens for Initial State
3  else
4  |  h_L ← IndividualHeuristic(S, S.pre.φ_i, S.pre.G_i, Π)
5  if h_L ≠ 0 ∧ h_L ≠ ∞ then
6  |  ⟨h_G, φ_i, G_i⟩ ← S.pre.⟨h_G, φ_i, G_i⟩      // h_G, φ_i, G_i remain unchanged.
7  else
8  |  if G achieved then
9  |  |  return ⟨0, ∅, ∅, 0⟩                          // Goal reached!
10 |  ⟨h_G, φ_i, G_i⟩ ← GlobalHeuristic(S, G, Π)        // See p.84.
11 |  h_L ← IndividualHeuristic(S, φ_i, G_i, Π)        // h_L is updated.
12 return ⟨h_G, φ_i, G_i, h_L⟩
```

4.2.2 Heuristic Calculation

Pseudocode for the heuristic calculation is shown in Function CalculateHeuristic. This simply updates h_L based on the IndividualHeuristic calculation, or, at a coordination point, performs the GlobalHeuristic calculation and updates h_G, ϕ_i, and G_i as well. A coordination point is defined as either the initial state (lines 1-2), any state where the current agent has completed all its goals (in which case $h_L \equiv 0$ in line 5), or the current agent is at a local dead end (in which case $h_L \equiv \infty$ in line 5).

A coordination point means that the GlobalHeuristic function is called which updates the global heuristic value h_G as well as the current agent ϕ_i and the set of subgoals G_i for that agent to achieve. The process for this method is described in the next section. Whether or not the current state is a coordination point, the individual heuristic value for the current agent, dependent on its subgoals, is calculated (line 4 or line 11). The sum of h_L and h_G will be used as the heuristic value of the state in determining the order in which it is expanded in the best-first search.

4.2.3 The Individual Heuristic Calculation

The calculation of h_L is similar to the calculation of the heuristic value for a state used by FF. This was discussed in Section 2.1.4.1 and will not be repeated here. The only difference is that the problem only considers agent ϕ_i's subproblem along with the goals

G_i which have been carried over from the last coordination point.

Definition The agent subproblem for agent ϕ_i of separated AVD $\Pi = \{V, \Phi, I, G, X, A\}$ with subgoals G_i is the MPT $\Pi_i = \{V_i, I_i, G_i, X_i, A_i\}$, where each element of Π_i (except G_i) is equal to the equivalent element of Π restricted to only contain elements that do not contain any variables from $V \setminus (\phi_i \cup P)$.

In other words, an agent subproblem is the full planning problem restricted so that it does not contain any elements that are internal to any of the other agents in the domain. This will always be smaller than the original problem for MPT's with more than one agent, which generally means that the subproblems can be solved quickly in relation to the overall problem. As an example, in the Robots problem, agent a's subproblem is the problem without any other agent's actions or the variables representing their locations or whether or not they have reported; however, it does include which locations are free. Note that the subgoal calculation (introduced later) will guarantee that the elements of G_i all belong to $P \cup \phi_i$.

4.2.4 The Global Heuristic Calculation

The global heuristic calculation returns the global heuristic value (h_G), the next agent to be used to calculate the local heuristic value (ϕ_i), and a set of subgoals for that agent (G_i). The aim of finding subgoals is to find a set of propositions that can be reached using only a single agent subproblem (i.e. in round 1 of relaxed plan generation) and that are a good stepping stone towards solving the overall planning problem. Pseudocode for this method is shown in Function GlobalHeuristic.

This process is split into two main parts: The first part is relaxed plan generation (lines 1–10). In this part, each agent generates their own relaxed planning graph from the current state, the final states are shared, and the process is repeated from the combined final states until all goal propositions have been reached. The second part (lines 11–25) involves choosing the next local agent and finding its subgoals that are to be achieved. In this part, relaxed plans are extracted in order to find out which propositions need to be achieved to traverse between agents' individual problems.

4.2.4.1 Generating Relaxed Planning Graphs

Generating relaxed planning graphs at coordination points relies on the relaxed planning graph generation methods presented in Chapter 2.1.4.1. The relaxed planning graphs

Function GlobalHeuristic(S, G, $\Pi = \langle V, \Phi, I, G, X, A \rangle$)

```
1  goals_left ← |{g ∈ G | g ∉ S}|                          // Used for h_G.
2  foreach a ∈ A, v ∈ V, φ ∈ Φ do
3  |  Reset internal values.    // Only happens once at start, see p.91.
4  r ← 0                                  // Count the number of rounds.
5  repeat
6  |  r++
7  |  foreach φ_i ∈ Φ do
8  |  |  RPG_{i,r} ← GenRelaxedPlanningGraph(S, r, Π_i)      // See p.91.
9  |  S ← reached(RPG_{1,r}) ∪ ... ∪ reached(RPG_{n,r})   // Assuming |Φ| = n.
10 until G all achieved in S OR no new propositions added
11 if G all achieved in S then
12 |  h_G ← M × r + N × goals_left                  // See Section 4.2.4.3.
13 |  goals ← ∅                          // The following finds φ_i and G_i.
14 |  foreach g ∈ G do
15 |  |  if g.r = 1 then
16 |  |  |  goals.add(g)      // If g first added in round 1 use it.
17 |  |  else
18 |  |  |  goals.add(ExtractRelaxedPlan(g))  // Else extract to find
   |  |  |                  // subgoals from round 1 (Section 4.2.4.2).
19 |  foreach s ∈ goals do
20 |  |  i ← s.agent     // The agent that achieved s at lowest cost.
21 |  |  φ_i.G.add(s)
22 |  φ ← arg max_{φ_i}(|φ_i.G|)                  // Ties are broken randomly.
23 |  G_i ← φ.G
24 else
25 |  return ⟨∞, ∅, ∅⟩              // Global Dead End (Corollary 4.2.2).
26 return ⟨h_G, φ, G_i⟩
```

are created by exactly the same method as that used in FF, except that they are only ever created for agent subproblems (as introduced in the previous section). That is, each agent applies the 'no delete lists' heuristic and creates a graph by adding *their* applicable actions repeatedly and updating the state until they cannot perform any new actions.

The first two lines of the function are used for reseting the relaxed planning graph values stored in the actions and propositions. This sets their related *cost*, *achieved_by*, and agent values to null, the purpose of which is discussed at the end of this section. However, for now, it should be noted that this happens once, at the beginning of the algorithm, and does not occur in between the creation of relaxed planning graphs.

To give an example, Figure 4.4 shows the relaxed planning graphs of each agent's subproblem for the domain instance shown in Figure 4.8, which can be compared to the full planning graph for the problem which was shown in Figure 2.2 on page 24. It can be seen that the separated planning graphs contain fewer actions in total than the collected graph for the full problem. This is because there are a lot of propositions that are not achievable by an individual agent working alone. For example, agent *b*, that corresponds to the agent with robot *b*'s location in its internal variables, cannot perform any actions at all because it is surrounded by other robots and does not have any access to those agents' actions.

It can also be seen from the figure that the starting states for each agent are much smaller than the starting state in the full problem. This is because each agent only has access to the public variables and their internal variables. For example, from agent *a*'s perspective, the proposition `at(B, z)` is not part of the problem. This means that creating individual planning graphs is simpler than creating planning graphs for the full problem.

The one downside of creating individual planning problems is that not all states reachable in the full problem are still reachable in the individual graphs. For example, in the full relaxed planning graph for the single-agent version of the problem, the goal state is eventually reached; however, in the individual graphs, only agent *A* can reach its associated goal. In order to make sure that every state reachable in the full problem is still achievable, a method needs to be put in place for sending information between the agents. A first attempt may be to broadcast public information that an agent adds to the graph at the time that it's added, but this leads to a huge communication overhead and ends up just being a convoluted way of creating the graph for the full non-decomposed problem.

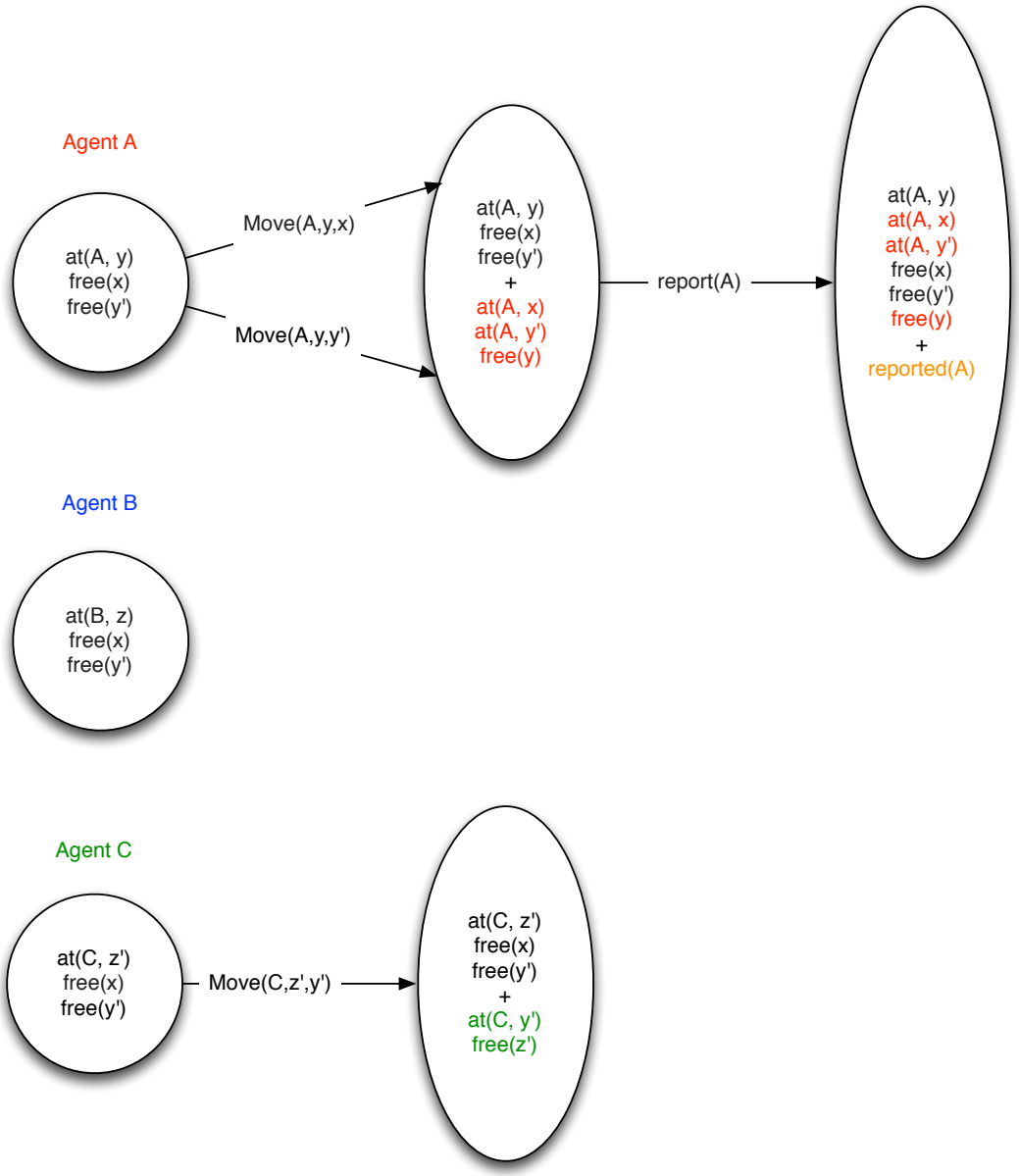

Figure 4.4: The relaxed planning graphs of the three agents from the starting state of the problem shown in Figure 3.1 on page 50. The propositions are colour-coded by the agent that first added them, propositions from the initial state are black. The orange text shows a goal atom. Agent B cannot perform any actions.

A much better solution is to take the collected final state of the relaxed planning graphs (line 8 of the pseudocode) and use this as the starting point of another round of relaxed planning graph generation. Due to implementation issues that will be discussed below, this operation can be performed very efficiently while restoring the property that all reachable propositions will at some point be reached using this process.

Proposition 4.2.1 *The above mentioned method of repeatedly generating relaxed planning graphs for each agent from the new state formed by the collected reachable propositions of all previous planning graphs will always terminate by finding all goal propositions if this is possible in the full problem.*

Proof To show this, an equivalent claim is proven, that all states reachable in the full problem are reachable via this method. This is equivalent because the goal propositions can be any propositions from the full problem. This is a well known property of relaxed planning graphs for the full problem, which comes from the fact that the relaxation is a strict weakening of the problem.

So, all that is needed to be shown is that the combination of the agent subproblem planning graphs is equal to the full relaxed planning graph. Firstly, all actions in the full problem belong to at least one agent in the decomposed problem, this was shown in Corollary 3.3.1 on page 57. Secondly, because we are dealing with separated AVDs, there are no joint actions, so ever action that belongs to an agent can be performed using only that agent's subproblem. Thirdly, any public proposition reached in the subproblems will later be added to every agent's relaxed planning graphs.

Assume, by way of contradiction, that p is a proposition reachable in the full planning graph but not reachable by Algorithm GlobalHeuristic. Then, there must be some action a that added p that can be performed in the full problem but can not be performed in any of the agent's subproblems at any point in the algorithm. The reason that the action can not be performed must be that one of it's preconditions, call it p', is not reached in the subproblems. However, there must be some action a' that added p' in the full problem. This line of reasoning can be continued to p'', a'' etc. until eventually an action a^* that can be performed from the initial state (if no such action exists, then the full relaxed planning graph must be empty). But it must be possible to perform this action in one of the agent's subproblems due to the properties mentioned above (no joint actions) and then a route can be found back to the original proposition p that shows it is in fact reachable. □

A simple corollary of this proof

Corollary 4.2.2 *If the multiagent relaxed plan generation method does not find all the elements of a particular goal set, then that goal set is unreachable in the full problem.*

Note that the reverse implication is not true because under the relaxation there may be reachable states that are not reachable in the full problem due to the presence of delete lists. This result is exactly the same as for relaxed plan generation for the full domain.

Figure 4.5 shows an example of the second layer of relaxed planning graphs, formed after the creation of a new starting state from the union of the end states of the first round of planning graphs (shown in Figure 4.4). Note that because each agent's subproblem only contains public and internal variables, propositions that belong to internal variables of other agents are effectively not distributed. In this second round of relaxed planning graphs, agent *b* can start performing actions by using the propositions `free(y)` and `free(z')` that it received from agents A and C respectively. As can be seen in the figure, both agent *B* and *C* can use the fact that location *y* is now free to eventually report at the goal location. It is common that goal states become reachable very quickly, usually in the second round of planning graphs, in fact, in all the problems from the IPC domains, it never took more than 3 rounds of creating agent planning graphs for all goals to be reachable. This is due to the power of the 'no delete lists' heuristic.

In terms of the actual implementation of ADP, the planning graphs are not literally created. Instead, propositions are appended with an estimated cost, an achieving action and related agent depending on which agent adds the proposition at the lowest estimated cost. This information will be useful for the next step, where plans are extracted from the planning graphs. The propositions in the domain are split up into the set of public propositions and internal propositions as shown in Figure 4.6.

Figure 4.6 also shows the estimated costs and achieving actions for each proposition at the end of the generation of relaxed planning graphs. For example, the proposition `reported(B)` has cost 4 because it required *A* to move, *B* to move twice, and *B* to report in order to be added. The achieving action is `report(B,x)` as it was this action that added the proposition `reported(B)`.

Function GenRelaxedPlanningGraph shows how the relaxed planning graphs are generated. This is similar to the standard implementation for generating relaxed planning graphs, however, it can be seen from the pseudocode the adjustments needed when working with multiple agents. This amounts to recording a best achieving agent for each proposition (which is overwritten if another agent adds the same proposition at a lower estimated cost) and also the round in which a proposition first appears.

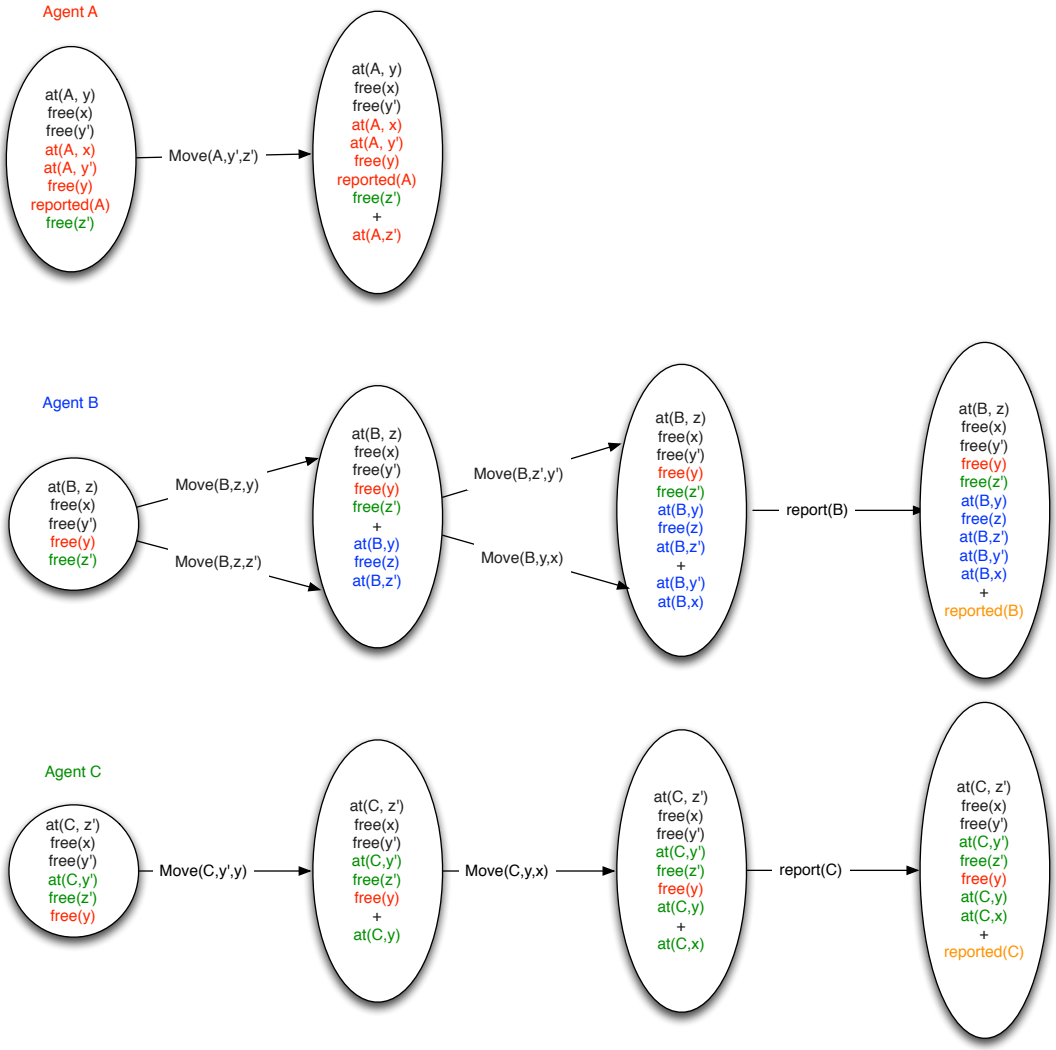

Figure 4.5: The relaxed planning graphs of the three agents from the collected final states of the previous planning graphs. Now agent C can achieve its goal. Agent B still cannot, but will be able to in the next iteration of this process. The propositions are colour-coded by the agent that first added them, propositions from the initial state are black.

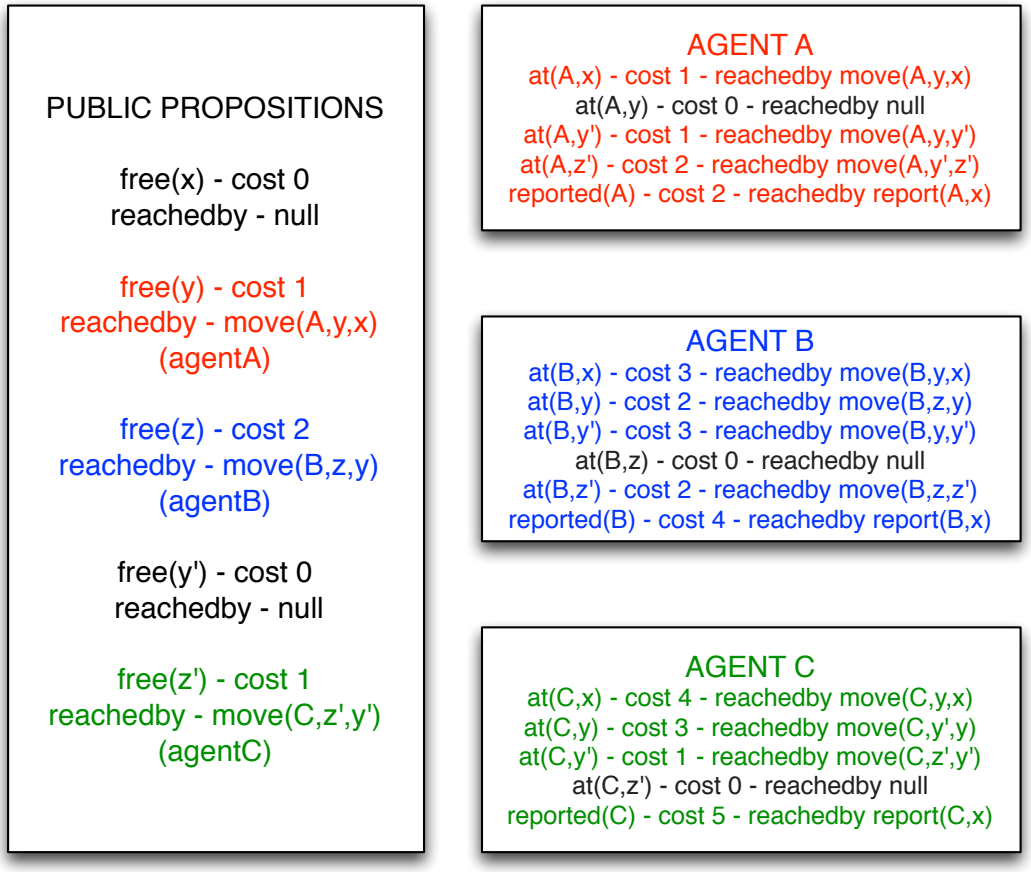

Figure 4.6: All the propositions for the Robots example problem after the creation of relaxed planning graphs step has finished. The propositions in black text represent those present in the initial state.

Function GenRelaxedPlanningGraph(S, r, Π_i)

```
1  foreach a ∈ A_i do
2      a.unsatisfied_preconditions ← |pre(a)|
3  queue = []
4  queue.addall(S)
5  while queue ≠ [] do
6      p ← queue.pop()
7      foreach a ∈ A_i such that p ∈ pre(a) do
8          a.cost ← a.cost + p.cost
9          a.unsatisfied_preconditions ← a.unsatisfied_preconditions − 1
10         if a.unsatisfied_preconditions = 0      // All preconditions met.
11         then
12             foreach p' ∈ eff(a) do
13                 if r < p'.r OR (r = p'.r AND a.cost < p'.cost) then
14                     p'.r ← r
15                     p'.cost ← a.cost
16                     p'.achieved_by = a
17                     p'.agent = φ_i
18                     queue.add(p')      // Only if not in queue before.
```

Note that the best costs, achieving actions and agents are not reset in between each individual call of the function. They are only reset once at the beginning of the whole process (lines 2-4 in Function GlobalHeuristic). This is because each round of relaxed planning graphs builds on the propositions found in the previous rounds. The only internal book-keeping values that are reset each time the GenRelaxedPlanningGraph method is called are the counter for the number of unsatisfied preconditions of actions (lines 1–2). For actions that were applied in a previous round, these will have already reached 0 again once all the elements of S have been processed from the Queue.

4.2.4.2 Extracting Relaxed Plans

Once the planning graphs have been generated, the next step involves extracting relaxed plans to find useful propositions that can be reached by a single agent from the current state. Note that there are many domains, such as Rovers, for which this process is never needed. In Rovers, all goal propositions are reachable by an individual agent from

any state that appears during search. When it is needed, plan extraction proceeds as described in Section 2.1.4.1, by picking a goal proposition, and then working backwards until the initial state is reached. The difference is that whenever plan extraction adds a proposition p such that $p.r \equiv 1$ and that the previous proposition p' had $p'.r > 1$, then p is added to the set of subgoals to be returned. As such, the ExtractRelaxedPlan method returns a set of propositions (all achievable in round 1) that all appear in the relaxed plan to achieve the current goal proposition for which a plan is being extracted.

Figure 4.7 depicts the extraction process for agent B's relaxed planning graph (originally shown in Figure 4.5). The queue starts containing the goal propositions, because `reported(B)` is a goal state, so that the extraction starts there. It then finds that the action that achieved `reported(B)` was `report(B)`, which will be linked by the propositions achievedby field in the implementation. The preconditions of `report(B)` are then added to the queue to be extracted from, and, in turn, the action that achieved them is found. This process repeats until all propositions in the queue are in the current starting state.

For example, for the `reported(B)` proposition the relaxed plan would be:

```
move(A,y,y'), move(B,z,y), move(B,y,x), report(B,x).
```

The important proposition here was `free(y)`, which was added in round 1 (by agent A's action `move(A,y,y')`) and allowed for the round 2 proposition `at(B, y)` to be achieved. The proposition `free(y)` is therefore returned as a subgoal.

4.2.4.3 The Next Agent, Goal Set and the Global Heuristic Value

Coming to the final part of the GlobalHeuristic function, the algorithm returns the next agent, its goal set, and the global heuristic value. Each subgoal is given to the agent that achieved it at the lowest cost during relaxed planning graph generation (it's h_add value), then, the agent with the most goals assigned to it is chosen as ϕ_i and its goal set used as G_i (lines 19–23). For ties, the agent is chosen arbitrarily.

Finally, the global heuristic value is calculated as

$$h_G = M * r + N * goals\_left.$$

Where r is the number of rounds of relaxed planning graphs that were created, and $goals\_left$ is the number of goal propositions that remain to be solved in the current state. M and N are some large numbers with $M > |G| * N$ and $N > \max h_L$. This means that the search prefers states with the lowest number of planning rounds required and, if this is equal, the lowest number of goals left to be achieved.

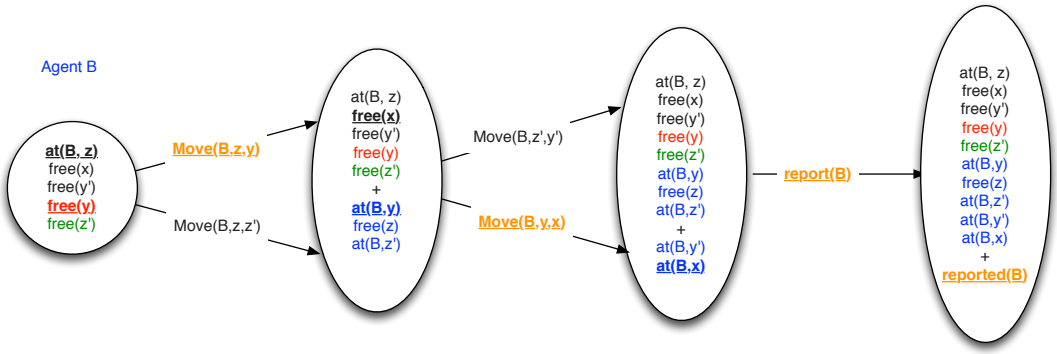

Figure 4.7: This figure shows the route back through the relaxed planning graph to extract from the goal proposition `reported(B)`. The actions used are in orange, while the propositions are bold and underlined. The extraction ends up requiring `free(y)` which it can be seen was added by agent A. The extraction then goes back through agent $A's$ graph from this point until it reaches the initial state.

4.2.5 Solving the Robots Domain

This section provides a walkthrough of ADP for the problem shown in Figure 4.8. The problem is assumed to come with the obvious agent decomposition that separates the robots from one another (which is exactly the decomposition returned by the algorithm presented in the previous section). As with the other examples presented in this thesis, the solution is very obvious to a human and consists of treating the robots as separate entities; however, it contains coordination elements that make it tricky for a multiagent approach. The goal for the problem is `(reported a)` $\wedge$ `(reported b)`. Interaction is required because once robot a has reported to the starred location it must move out of the way in order for robot b to be able to reach that location.

The initial state is treated as a coordination point, so each agent would create their full relaxed planning graphs (see Chapter 2.1.4.1). For robot a, this would contain the atom `(reported a)`, a goal atom as this is reachable by a individually. During plan extraction it would also be found that a needs to add the atom `(free y)` in order for b to be able to reach the goal. As a has the most (only) goals in the first layer of planning graphs, it is chosen as the next agent with the subgoals `(reported a)` and `(free y)`.

The individual planning problem for a to achieve `(reported a)` and `(free y)` is worked towards by updating the local heuristic value resulting in the state shown in Figure 4.9. All the subgoals are achieved so this is another coordination point. As the full goal has not been reached, both agents create their relaxed plans again from the

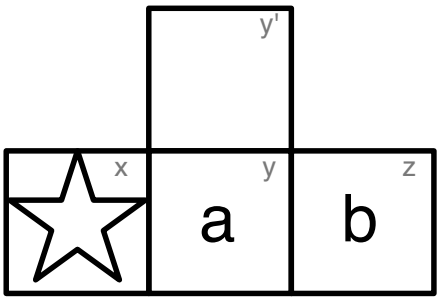

Figure 4.8: A variation of the Robots domain. Each robot (a, b) must report to the starred square. Robots can not move through one another.

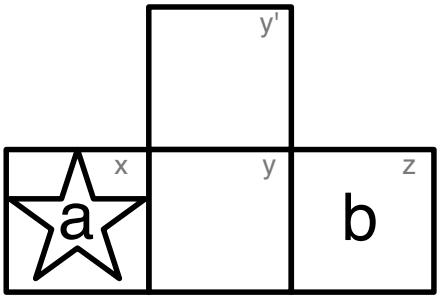

Figure 4.9: Robot *a* has reached its associated goal but b still cannot find an individual plan to get there.

current state. This time however, no new goal atoms are found in either of the individual planning graphs because robot *a* cannot complete `(reported b)` (as this is a value of an internal variable of robot *b*) and robot *b* cannot complete it as robot *a* is occupying the report location.

As no new goal atom has been reached in the first round of creating internal planning graphs, a new state is formed from the collected final state of each agent as depicted in Figure 4.10. This state is necessarily larger than (or the same size as) the starting state due to the 'no delete lists' relaxation that is being employed. Note that the state also contains `(free ?loc)` for each possible location as these are added for locations `x` and `y` when those agents move out of their starting locations.

From this new state it is possible for robot *b* to reach `(reported b)`; however, it still needs robot *a* to move out of the way first. Plan extraction works its way back

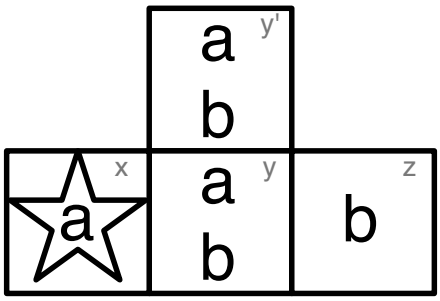

Figure 4.10: A representation of the state of all reachable propositions from each agent. This includes the atom `(free ?loc)` for each possible location.

through robot *b*'s second planning graph and finds that the atom `(free x)` is necessary in order to achieve `reported b`, and that it was robot *a* that added this. `(free x)` is then added as a subgoal for agent *a* to achieve, as this is reachable by a single agent from the current state.

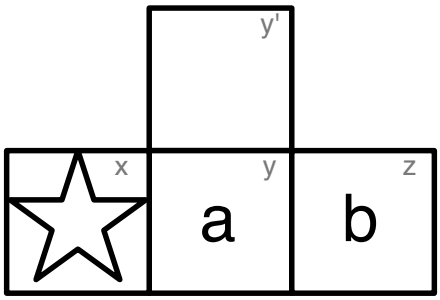

Figure 4.11: Robot *a* has moved out of the way for robot *b* but has ended up still blocking its route.

Unfortunately, the search process is not yet over, as robot *a* will solve its subproblem by moving to location `y` as shown in Figure 4.11, meaning that Robot *b* still cannot get to the goal location. However, the final plan is eventually found by repeating the process outlined above. The plan for *a* to move back left again leads to a state in the *closed* list and which will not be re-evaluated so the position shown in Figure 4.12 will be the next destination of the search. From here, it is easy to see how robot *b* will be assigned the subgoal to report at its goal location and then a full plan will have been found.

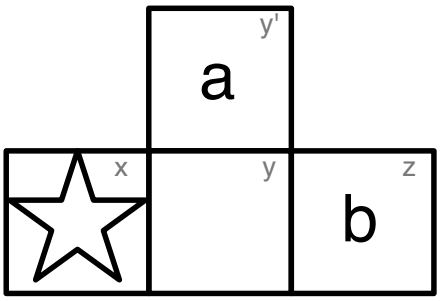

Figure 4.12: Robot b can finally reach the goal location.

The process outlined in this section follows the route that ADP takes through the search space, which is very similar to the human method for solving the problem. The problem is first split up into multiple agents and the easiest subgoals are planned for. If coordination is required between the agents then subgoals are found that help them achieve the global goal.

4.3 Summary

This chapter described an automated decomposition algorithm along with a plan search algorithm for multiagent classical planning problems. The automated decomposition algorithm takes a standard classical planning problem and returns a separated AVD for it, if one can be found. The generated decompositions are analysed in the evaluation chapter and shown to coincide strongly with the multiagent decompositions that a human expert would expect the problem to have.

The decomposition algorithm can be combined with ADP to provide a complete planner that takes standard input used by the planning community. In cases where a separated AVD is found, ADP is used, in other cases, a standard single-agent planner is invoked. Like other heuristic planning approaches ADP makes no guarantees as to the quality (in cost) of the returned plan; however, it is empirically shown to perform faster than state-of-the-art single-agent planners, on problems with inherent multiagent structure and returns plans of a reasonable cost. The evaluation of this is presented in the following chapter.

Chapter 5

Evaluation

This chapter presents the evaluation of the multiagent classical planning algorithms introduced in this thesis. There are two main sections, the first of which looks in detail at the decomposition algorithm and discusses the properties of the returned decompositions over different domains. The second main section presents the evaluation of ADP, comparing it to state-of-the-art single-agent planners and discussing how the composition of the decompositions effects performance. However, first, the details of the experiments are discussed.

5.1 Experiment Design

Both the decomposition algorithm and ADP were implemented in C++ as an extension to the Fast Downward (FD) planning system (Helmert (2006)). FD was used to parse the problem definitions and create the MPT representation of the problem, which was then used as input for the agent decomposition algorithm. Using FD as the underlying planner means that the input can be any problem written in propositional PDDL2.2, in other words, STRIPS domains with arbitrary propositional formulae in preconditions and goal conditions, and conditional as well as universally quantified axioms and effects.

FD calculates the MPT representation (Section 2.1.3) of the planning problem and also generates the causal graph (Section 2.1.4.2), used in the agent decomposition algorithm. The causal graph generation code was modified to exclude nodes (v, v') and (v', v) that are induced by the same action for use with the agent decomposition algorithm. When running the competing planning approaches the causal graph code was left unmodified.

It may be the case that there are multiple MPT representations of a given problem

and they will lead to different agent decompositions. The method implemented by FD attempts to find decompositions that maximise the size of variables while minimising their number, which proves to be a sensible heuristic for associating agents with internal variables that represent their state. MPT's were generated once for each problem and then used as input for each of the algorithms tested, in order to ensure that they remained constant for the different algorithms tested.

Direct comparison with other multiagent approaches is problematic as there are none competitive with ADP in terms of performance or breadth of the domains covered. That is why the aim of the ADP evaluation is to show that a multiagent approach can significantly improve performance over state-of-the-art single-agent planning problems. ADP is compared to single-agent algorithms that have shown good performance in the international planning competitions and that are also implemented in FD, a competitive and recently popular planning system with strong performances in the International Planning Competitions. This allows for a direct comparison that can focus on the performance of the algorithms themselves while minimising the effects of differing implementations.

All experiments were run on the same machine, a Dell Poweredge R610 with 48GB of memory and 2.66GHz processor. Each planner was allowed five minutes to return a solution on each problem. This time limit was chosen due to the large number of problems over which the algorithms were run and, as will be shown below, because it is long enough to cover the vast majority of the problem instances tested. The problems tested are ordered by the problem number in which they appear in the IPC problem sets. They generally get larger and more complex as the problem number increases but this is not always strictly true. As there is no direct correlation between the problem numbers and the properties of the problems, the numbers are sometimes omitted from the results. However, the ordering of the problems is always preserved.

The presented results include only those for which all tested algorithms took longer than 0.1 seconds to return a solution, the simpler problem instances being omitted as there is no way of drawing any sensible conclusions when the planning times are so close together. Solutions were also omitted for domains where no planner returned a solution within the time limit. This only occurs in a few problems across all the domains tested. All totals presented in tables sum over only the problems for which all the tested planners returned a solution within the time frame.

5.2 Automated Agent Decomposition Results

This section presents the empirical evaluation of the automated agent decomposition algorithm introduced in Section 4.1. The algorithm was run on every classical planning domain from the collected International Planning Competition problem sets.

The hypothesis under consideration was that:

> **Hypothesis 1:** It is possible to design a domain independent algorithm that can find the 'multiagent structure' of classical planning problems which can then be exploited for faster planning with a multiagent search algorithm.

It is impossible to corroborate this hypothesis without the results of ADP on the decomposed domains. This section therefore discusses the type of decompositions found and the way in which they can be considered to be 'multiagent'. The algorithm tested is, by design, domain independent and can be run on all the classical planning problems from the IPC benchmarks.

5.2.1 Decomposition Structure

Table 5.1 shows the collected results of the experiments, with a row for each domain that returned a decomposition. Some domains have multiple problem sets for different versions of the problem. For example, optimisation problems are designed for planners that return the lowest cost solution, whereas satisficing problems are for use with planners that are only concerned with returning a plan, regardless of cost. As ADP is a heuristic-based satisficing planner, when there is a choice, the results for the satisficing problems are presented. For further cases where there was a choice between problem sets, the most recent, non-ADL version of the problem were chosen. In practice, the returned decompositions generally did not vary in content between the different problem sets and the table is representative of all versions of the problem. The only slight exceptions are the Logistics and Satellite domains, whose results are split into two subsets based on the different problem sets; this is discussed in more detail in the appropriate sections.

There are fifteen different domains represented in the table out of a total of thirty six different domains tested, a decomposition was therefore found for around two fifths of all domains tested. It cannot be expected that all domains return an agent decomposition, of course, single-agent domains should ideally return no decomposition at all and there is no *a priori* reason for the IPC domains to be inherently multiagent. The fact that 40% of them do have a multiagent structure shows how prevalent multiagent domains are

Domain	Decompositions Found	No. of Agents Min	No. of Agents Max	Agent Vars (%)	Action Breakdown (%) Internal	Action Breakdown (%) Public	Internal Action Composition (%) I	Internal Action Composition (%) $>I$	Internal Action Composition (%) $I<$	Internal Action Composition (%) $>I<$
Airport	42/50	2	15	0.5	100	0	0	9.7	0	90.3
Depot	22/22	2	6	6	8.4	91.6	10.5	0	0	89.5
Driverlog	20/20	2	6	22.3	90.4	9.6	0	42.7	0	57.3
Elevators	20/20	4	8	14.4	100	0	8.7	0	0	91.3
Floortile	16/20	2	3	13.3	100	0	0.5	24.7	0	74.9
Logistics00	28/28	3	7	33.6	100	0	7.9	0	0	92.1
Logistics98	35/35	7	108	74.7	100	0	26.4	0	0	73.6
Mystery	31/35	2	16	81.4	100	0	49.2	0	0	50.8
Pathways	28/30	2	38	37.9	53.3	46.7	39.1	11.4	14.2	35.2
Rovers	38/40	2	14	58	100	0	69.6	0	7	23.4
Satellite	18/20	2	12	72.1	100	0	97.1	0	2.9	0
SatelliteHC	14/16	5	15	37.9	100	0	99.4	0	0.6	0
Storage	25/30	2	5	6.6	100	0	0.3	1.2	0.3	98.2
Tpp	26/30	2	8	28.9	100	0	9.4	0	0	90.6
Transport	20/20	4	4	17.2	100	0	8.8	0	0	91.2
Woodworking	20/20	2	15	15.7	44.8	55.2	22.5	0.1	0.1	77.4
Zenotravel	18/20	2	5	24.4	100	0	28.5	0	0	71.5

Table 5.1: Decomposition results for IPC domains that returned agent decompositions. Agent Vars % is the percentage of variables in the domain that are in an agent's variable set. I represents completely internal actions while the shorthand $>$ means influenced and $<$ means influencing. The $>I<$ column therefore lists the percentage of internal actions that are both influenced and influencing.

in the planning community and therefore the relevance of a multiagent planning even when discussing domains that were not specifically designed as multiagent.

A brief look at the table shows that it contains all the domains traditionally used in the multiagent planning literature (see Section 2.3), with Airport, Logistics, Rovers, Satellite, Transport and Zenotravel all making an appearance. Indeed, the algorithm returns a decomposition for all these domains and, it will be shown below, this decomposition corresponds strongly with the expected multiagent decomposition in all cases. This is the first positive result; that the algorithm returns expected decompositions in the most easily identifiable multiagent domains.

The first column of the table, 'Decomps Found', shows the number of problem instances for which a decomposition was returned out of the number of problem instances that exist for that domain. It can be seen that, when a domain is decomposable, a decomposition is returned for most of the problem instances. The missing decompositions are due to single-agent instances of the generally multiagent domains. For example, the Rovers domain contains forty problem files, but the first two of these only contain one rover object and are therefore single-agent meaning that they do not return a decomposition. That no decomposition is returned in these cases is a strong result towards the ability of the decomposition algorithm to pick out agents as we understand them. It shows that it is not returning false positives even in otherwise decomposable domains once the agents are removed.

The next set of columns in the table show the minimum, maximum and mean number of agents found for each domain. The results show a large number of domains that have just two agents as a minimum and seven or more agents for the more complicated versions, while there is only one domain for which the number of agents is the same in every problem instance. This is because the IPC problem domains are designed to contain problems of varying difficulty, with a problem set generally consisting of a single domain file along with multiple problem instances that vary in size. A common way to increase the size of a problem instance is to add more objects to the domain; and, more often than not, these extra objects correspond to the agents that the decompositions find. This facet of the results provides a further argument for the suitability of a multiagent approach to planning as it shows that there are many domains written with a multiagent structure and that this structure is a key element of how the domain is designed and extended. As domains get more complicated it is likely that they will contain more and more agents, suggesting that planning techniques focussed on exploiting multiagent structure will become more useful as the research area grows.

The 'Agent Vars (%)' column in the table shows the percentage of variables that belong to agents in the returned decompositions. It can be seen that this number varies wildly between the different domains, from just 0.5% in the Airport domain to 81% in the Mystery domain. Naively, it may be assumed that domains that assign more variables to agents will be easier to solve as they leave less interaction due to containing less variables for modelling the environment that the agents are acting upon. However, the percentage of agent variables does not appear to have any direct correlation with the other columns in the table. Looking at the Airport domain, with just 0.5% of the variables belonging to agents, it is still the case that all the actions in the domain are internal. Furthermore, it will be shown later that the percentage of agent variables does not seem to affect the performance of ADP. This means that, on the one hand, an agent decomposition can effectively decompose a domain by picking out a few variables that are enough to expose the underlying structure, while, on the other hand, there are domains for which most of the variables represent agents' internal states. Another way of looking at this result is that, under well-structured decompositions, at least 20% of the domain, and as much as 99.5% of the domain models the environment that the agents are acting in and not the agents themselves. This reinforces the importance of switching to a view of multiagent decompositions that includes an environment model as depicted in Figure 3.3 on page 53. It also highlights that it is not the placement of the variables that is important, but the partitioning of the actions that this induces.

The final set of columns in the table show which action types are found by the decomposition algorithm. The actions are split into the different types of internal actions, based on the influenced ($>$) and influencing ($<$) distinction, and the public actions. As only separated agent variable decompositions are found there are no joint actions under any of these decompositions. There are only four domains from that contain public actions and, it will be seen below, it is much harder to find plans in these domains. This is because the public actions belong to every agent in the domain, the more public actions in a decomposition, the less likely it is to be useful for solving with a multiagent approach. The results concerning the breakdown of the internal actions will be discussed when the domains are explored individually.

5.2.2 Decomposition Descriptions

Ideally, the decomposition algorithm would be expected to only return decompositions that are identifiable as multiagent, and which are beneficial to a multiagent search

Domain	Decomposition Description
Airport	The airplanes.
Depot	The trucks.
Driverlog	The trucks.
Elevators	The lifts.
Floortile	Robot's locations and the color they have access to.
Logistics	The vehicles; be it an airplane or a truck.
Mystery	Different 'craves' variables.
Pathways	Random looking collections of 'chosen'/'available' predicates of varying size.
Rovers	A set for each rover that also contains associated cameras and objectives.
Satellite	A set for each satellite that also contains associated instruments and calibrations.
Storage	The hoists.
Tpp	The trucks.
Transport	The trucks.
Woodworking	The surface-condition of different parts (but not all of them) and separate saws.
Zenotravel	The planes.

Table 5.2: Table describing the variables which make up the agent sets in the domain decompositions.

algorithm. In other words, to find multiagent decompositions for domains where such a decomposition exists, but not return any decomposition in the domains for which there is none. The algorithm certainly succeeds on the first point; however, for the second point, the returned decompositions need to be analysed in more detail. Table 5.2 shows the decomposition breakdowns for each domain. The descriptions in the table were created from manually reviewing the contents of the returned agent variable sets. The defining characteristic of the variables was picked out and used to describe the decomposition. For example, in the Driverlog domain, the agent variables contain propositions that relate to the location of one of the truck objects. This variable is then said to be primarily about that truck. The decompositions are consistent enough across each domain, such that the description can be applied to each problem instance.

In all cases, except for Mystery, Pathways and Woodworking, the decomposition variables correspond to the expected decomposition. The decompositions generally pick out exactly the part of the domain to be expected and that is used in the multiagent planning literature, where applicable. In some cases, such as Floortile, Rovers and Satellites, the returned decompositions are more detailed than may have originally been expected, associating certain parts of the domain with the agents such as instruments they are carrying or certain properties that they have.

For the three domains with unintuitive decompositions, the algorithm returns a valid agent variable decomposition, yet not one that is easily understandable or identifiable as obviously multiagent. It is certainly possible to view these as multiagent domains with non-standard agents, and the nature of the decompositions along with the plan search results using these decompositions is discussed below.

Of the IPC domains that do not appear in the table, it could be imagined that some of them might have a multiagent decomposition, though in no case is it as obvious as for the domains for which a decomposition was found. For example, in the Freecell domain, a suit of cards has to be arranged, by performing a series of actions to move them that abide by certain rules. It could be imagined that the cards themselves correspond to separate agents with their location in the puzzle being their internal state. However, due to the way that the actions work in the domain, there is no factor of the card locations that can be identified as internal. A card's location is defined by the card that it is on top of, and therefore all the cards are inter-related and there is no variable that can model a single card's internal state. It should be noted that early version of the decomposition algorithm, before the link to internal states was realised and MMPTs formalised, would find varying possible decompositions for the Freecell domain, but never any that led to

improved planning times.

In summary, the decomposition algorithm finds the expected decomposition for the obviously multiagent domains. It also finds decompositions in other domains, and these are verified to be valid agent decompositions, though they do not correspond to decompositions that a human would make for the domain. The next section looks in more detail at the breakdowns of agents and actions under the returned decompositions for selected planning domains.

5.2.3 Decompositions by Domain

The most commonly used domains in the multiagent planning literature are Rovers, Satellites and Logistics. This is due to their obvious multiagent structure and the fact that, in the case of Rovers and Satellites, they are loosely coupled, while Logistics has many very small problem instances so is suitable for testing non-competitive planning implementations. In Rovers, multiple rover vehicles navigate a planet's surface in order to obtain and communicate soil and rock data. The natural decomposition, and the decomposition used in the multiagent planning literature, is to split the rovers up as separate agents. In the Satellites domain, multiple satellites need to calibrate their equipment and facing in order to take images of stars and planets. Naturally, in the multiagent literature, this domain is broken down by the separate satellites. Finally, in the Logistics domain a network of trucks and airplanes must deliver objects (that represent packages) to specified locations. In the multiagent literature this is broken down into separate agents for each vehicle, meaning that the agents can be either trucks or airplanes. The first part of the detailed empirical analysis will explore the decompositions found for these domains, where there is a well explored and understood multiagent decomposition.

5.2.3.1 Rovers

Table 5.3 shows the results of the decomposition algorithm over problems from the Rovers domain, although, for brevity, not all problem instances are included. Note that this table, and the rest of the tables presented in this section show the total number of actions under each category as opposed to the percentages. This is so that the size of the problems can be inferred from the table.

The first thing to note is that no agents were found for problems 1 and 2. Initially this looks problematic for the algorithm; however, looking into the problem files shows

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	0	–	–	–	–	–	–	–	–
2	0	–	–	–	–	–	–	–	–
3	2	7	2	30	0	23	0	0	7
4	2	7	2	31	0	25	0	0	6
5	2	19	3	84	0	69	0	15	0
6	2	25	1	148	0	138	0	10	0
7	3	7	11	117	0	59	0	6	52
8	4	16	13	189	0	87	0	32	70
9	4	23	8	225	0	177	0	24	24
10	4	18	19	257	0	125	0	26	106
15	4	19	12	325	0	240	0	25	60
20	8	41	35	1903	0	1218	0	208	477
25	10	41	17	2200	0	1406	0	260	534
30	10	66	44	5356	0	3718	0	378	1260
35	12	97	115	10572	0	7189	0	584	2799
40	14	190	117	24654	0	18215	0	1449	4990

Table 5.3: Table showing the agent decomposition results for the Rovers domain.

that these domains contain only one rover object so no decomposition is expected to be found. The problems are roughly ordered by complexity, and the first instances are small enough that they only include one rover. For these problems there is no obvious decomposition and it would be expected, as is indeed the case, that the algorithm does not return a decomposition either. Furthermore, testing has shown that artificially modifying any of the other problem files to have only one 'rover' object results in no decomposition being returned.

Problem 3 is the first to contain multiple rovers in the problem definition, exactly the same number as agents found by the decomposition algorithm. Looking into the results further, the number of agents found (shown in the second column of the table) always corresponds to the number of rovers in the problem file. Table 5.2 also shows that the decompositions correspond to the rovers in the domain.

The third column of the table shows that the number of variables in the agent variable sets is always greater than the number of agents. This is because the extension part of the decomposition algorithm is successfully managing to increase the size of the agent variable sets. If this was not the case, only singleton sets would be returned and the number of agent variables would be equal to the number of agents. Looking deeper into the returned decompositions shows that each agent always contains, as the first variable found in the decomposition, a variable representing the location of a particular rover. Then, depending on the problem, this is extended to include variables that represent whether or not that rover has calibrated its onboard cameras and whether or not it has collected certain images. These extra variables appear only if they are unique to the agent in question and, for some problem instances, there are rovers that could not be extended in this way. This is a strong result, as it shows that the extension part of the decomposition algorithm is working as intended, and that the algorithm can find more detailed decompositions than are created by humans.

The next column of the table shows the number of public variables for the domain. This is not directly relevant to the quality of the decomposition but is a good indicator of the structure of the problem under its decomposition. The number of public variables represents the size of the environment that the agents are acting in. In the rovers case, the number of public variables is only marginally less than the number of agent variables, meaning that roughly 60% of the domain is distributed by a multiagent approach.

The final columns of the table show the breakdown of actions in the domain and in most cases, as with in Rovers, all the actions are internal under the decomposition. This is expected in domains such as rovers that have a very intuitive multiagent decomposition

and are known to be loosely coupled. The internal actions are then broken down further into whether or not they are influenced or influencing. As was discussed in Chapter 3, these represent different complexities for multiagent planning. The rovers domain has a large number of completely internal actions which suggests that it should be very amenable to a multiagent planning approach.

In summary, the results for the Rovers domain conform to the human decompositions and even managed to provide more detailed decompositions including instruments when they are only relevant to a single rover. Furthermore, the algorithm does not return decompositions for domains with only one rover, which is a nice result as these domains appear to be very single-agent in nature and it would be problematic if an abnormal decomposition was returned in this case.

5.2.3.2 Satellites

The results of running the decomposition algorithm on the Satellites domain are shown in Table 5.4. The table is split into the standard satellite problems (top) and the hand-coded, generally larger and more complicated problems (bottom). The results contain the same property as for the rovers domain where the early problem instances do not return a decomposition. Again, this is because these problem instances only contain one of the objects that is related to the agents in the other versions of the problem.

The table shows a property of the Satellites domain that does not exist in the Rovers decompositions, namely, that there are no influenced actions in the domain. This can be seen because there are no actions in the columns for $> I$, $> I <$, or any public actions. This means that the agents can act completely independently, in fact, while they might need to alter the environment to achieve the goal, their possible actions do not require the environment to be in a specific state. This means that the ADP algorithm will never have to generate a second round of relaxed planning graphs and that it is impossible to make a wrong decision when choosing the agent to act next. From this it can be concluded that ADP, and other multiagent approaches, should be especially effective on the Satellites domain.

The decomposition variables show that, as would be expected, the decompositions found correspond to the satellites in the domain. As with the Rovers domain, there are more agent variables than there are agents showing that the decompositions contain more than one variable. Analysing the decompositions shows that not only are the satellites separated, but also the instruments that they are carrying, and whether or not they have power or are calibrated.

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	0	–	–	–	–	–	–	–	–
2	0	–	–	–	–	–	–	–	–
3	2	11	4	141	0	132	0	9	0
4	2	13	4	213	0	203	0	10	0
5	3	24	6	360	0	318	0	42	0
6	3	18	5	375	0	360	0	15	0
7	4	24	7	607	0	572	0	35	0
8	4	28	10	948	0	894	0	54	0
9	5	32	10	1162	0	1110	0	52	0
10	5	32	11	1472	0	1420	0	52	0
15	8	54	18	5005	0	4903	0	102	0
20	5	67	40	3346	0	3148	0	198	0
1	5	32	34	9370	0	9090	0	280	0
2	5	30	43	14110	0	13835	0	275	0
3	7	50	47	19909	0	19389	0	520	0
4	7	44	66	37642	0	36874	0	768	0
5	8	50	60	43001	0	42141	0	860	0
6	8	56	64	44025	0	43324	0	701	0
7	10	54	64	56097	0	55595	0	502	0
8	10	68	93	110461	0	109330	0	1131	0
9	15	86	87	164824	0	163955	0	869	0
10	15	92	108	233874	0	232670	0	1204	0
15	8	48	184	336040	0	334648	0	1392	0

Table 5.4: Table showing the agent decomposition results for the Satellites (top) and SatelliteHC (bottom) domains.

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	5	5	10	260	0	20	0	0	240
2	5	5	10	260	0	20	0	0	240
3	5	5	11	284	0	20	0	0	264
4	5	5	11	284	0	20	0	0	264
5	5	5	12	308	0	20	0	0	288
6	5	5	12	308	0	20	0	0	288
7	7	7	13	570	0	50	0	0	520
8	7	7	13	570	0	50	0	0	520
9	7	7	14	610	0	50	0	0	560
10	7	7	14	610	0	50	0	0	560
15	3	3	4	54	0	6	0	0	48
20	3	3	6	78	0	6	0	0	72
25	4	4	8	156	0	12	0	0	144

Table 5.5: Table showing the agent decomposition results for the Logistics domain.

It can be seen that there is a strong correlation between the breakdown of variables and actions and the domain under consideration, regardless of which problem instance is being analysed. This is because the domain files are invariant over these problems and they set up the general structure of the domain, while it is the problem files which introduce the agents themselves. However, one difference between the Satellites domain and other domains is that as the problems increase in complexity, the size of the public variable set grows in proportion to the number of agent variables. This is because the domains are not enlarged simply by adding more satellites. Instead, the complexity of the environment, the number of stars, planets and phenomenon is increased as well. This suggests that ADP will scale better on the Rovers domain than on the Satellites domain as it should cope well with the introduction of agents but a more complicated domain should hinder it just like it would a single-agent planning problem. However, perhaps the Satellite domain will be solved better initially because of the fact that it contains no influenced actions. The observations about the structure of the decompositions and how they might relate to the performance of ADP will be re-introduced in the relevant part of the ADP evaluation.

5.2.3.3 Logistics

The decomposition results for the Logistics domain are shown in Table 5.5. The table shows the results for the problems from the year 2000 version of the domain; however, the discussion applies equally to the 1998 version which has only minor differences in the domain definition and returns decompositions with similar structure. This domain is interesting because of the multiple ways by which it can be decomposed. The method used in the recent literature defines an agent for each vehicle in the problem, this bridges two different object types; trucks and planes.

Analysing the decomposition variables confirms that the returned decompositions contain an agent for each of the vehicles in each problem instance. Unlike both Rovers and Satellites, the agent sets in this domain are all singleton and the extension part of the algorithm fails to enlarge any of them. Looking into the domains in more details this also makes sense; there is no part of the domain that can be associated with just a single vehicle.

Even though the decompositions found clearly relate to what can be considered as agents, they contain a large number of both influenced and influencing actions. These are the hardest type of internal actions to deal with as they can affect, and be affected by, every other agent in the domain. It is natural that there will be multiagent domains that contain a lot of coordination between the agents and it is a positive result that the expected decompositions are found for these domains and shows that the idea of planning agents as entities with internal states extends to highly interactive domains. However, from this action breakdown, it is expected that ADP will do comparatively worse than in both Rovers and Logistics. However, it will hopefully still show improvements over the competing planners.

5.2.3.4 Airport

Moving further from the core domains used in multiagent planning, there are others that occasionally appear in the literature. Of these, the most interesting is the airport domain, for which Table 5.6 shows the decomposition results. For this domain there are many problems for which no decomposition is found; however, again, this corresponds to the problems in which there is only one agent. Looking into the results further, the decompositions returned always correspond to the airplanes referenced in the problem file.

Examining the problem files for the airpot domain further, it can be seen that they

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	0	–	–	–	–	–	–	–	–
2	0	–	–	–	–	–	–	–	–
3	2	2	44	242	0	0	23	0	219
4	0	–	–	–	–	–	–	–	–
5	0	–	–	–	–	–	–	–	–
6	2	2	112	683	0	0	75	0	608
7	2	2	112	683	0	0	75	0	608
8	3	3	150	1123	0	0	144	0	979
9	4	4	188	1388	0	0	150	0	1238
10	0	–	–	–	–	–	–	–	–
20	7	7	334	3161	0	0	346	0	2815
30	8	8	1762	23470	0	0	2792	0	20678
40	4	4	1273	15039	0	0	1522	0	13517
50	15	15	4078	58691	0	0	5602	0	53089

Table 5.6: Table showing the agent decomposition results for the Airport domain.

do not explicitly declare any objects. Instead, the objects are written into the actions and propositions of the domain, which is a valid, but uncommon way of specifying a planning problem in PDDL. It was argued in Chapter 3 that the decomposition algorithm's main motivation was a shift in focus from an action-centric view of agent decompositions to an object-centric view. However, the results for the airport domain show that the algorithm succeeds even without objects explicitly defined, due to analysing the MPT structure of the domain instead of a direct analysis of the objects or types in the domain.

The most interesting point to note about the Airport decompositions is that the ratio of agent variables to public variables is very low, lower than for any other domain. For example, in problem 40, just 4 of the 1,277 variables are assigned to agents. Even so, this variable decomposition induces a decomposition of the actions in the domain such that all 15,039 of them belong to an agent. It is remarkable that, as long as the correct variables are picked out, the agent structure of a domain can be defined with such a small number of variables. Naturally, with so many public variables, the majority of the actions in the domain are both influenced and influencing leading to a complicated decomposition that requires lots of coordination.

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	2	2	12	60	180	12	0	0	48
2	2	2	18	108	600	12	0	0	96
3	2	2	24	156	1260	12	0	0	144
4	2	2	30	204	2160	12	0	0	192
5	2	2	36	252	3300	12	0	0	240
6	2	2	51	372	7200	12	0	0	360
7	2	2	27	156	1620	12	0	0	144
8	2	2	39	252	3900	12	0	0	240
9	2	2	54	372	8100	12	0	0	360
10	2	2	30	348	3060	60	0	0	288
15	2	2	61	780	17850	60	0	0	720
20	4	4	63	2144	23550	224	0	0	1920

Table 5.7: Table showing the agent decomposition results for the Depot domain.

5.2.3.5 Depot

The Depot domain is the first that will be discussed that returns a decomposition with a large number of public actions, the results of which are shown in Table 5.7. These represent actions that do not belong to any of the agents. Instead, they directly manipulate the environment. The depot domain contains trucks and hoists that manipulate the locations of crates. The trucks are modelled so that they can be loaded (by the hoists) with the crates and then transport them to a different location, this means that the location of the trucks are valid agents. The hoists however, do not form valid agents as they directly change the locations of crates. There is no variable in the domain that can represent their internal state.

Considering the preceding, it is easy to understand why the decompositions contain so many public actions. In fact, this domain simply represents a type of multiagent problem with a complicated environment that dominates the small part of the domain that can be said to be multiagent. It is possible that ADP will not perform better than the single-agent planners over this domain. ADP cannot extract any benefit from the large amount of actions that solely manipulate the environment and it will be interesting to see if the extra structure and computation from using a multiagent approach is worthwhile compared to the small part of the structure that can potentially be exploited.

Prob.	No. of	Variables		Total Actions		Internal Action Breakdown			
No.	Agents	Agent	Public	Internal	Public	I	$>I$	$I<$	$>I<$
1	2	19	251	3679	5580	265	0	0	3414
2	3	26	266	4995	4796	443	0	0	4552
3	9	82	251	4751	5909	1887	0	0	2864
4	3	30	337	5692	7800	486	0	0	5206
5	4	39	352	5982	8478	804	0	0	5178
6	6	57	336	6370	7800	1240	0	0	5130
7	3	27	279	3394	6864	478	0	0	2916
8	6	56	266	4107	6842	1231	0	0	2876
9	6	54	247	4127	5890	1491	0	0	2636
10	3	15	0	258	0	258	0	0	0
11	10	93	312	6595	7536	2415	0	0	4180
15	9	86	412	9160	9672	2002	0	0	7158
20	5	21	295	6222	8392	558	27	27	5610

Table 5.8: Table showing the agent decomposition results for the Woodworking domain.

5.2.3.6 Woodworking

The final domain that will be discussed in detail is Woodworking, for which the results are shown in Table 5.8. This domain models a woodworking environment where different 'parts' have to be planed, ground, sawn and cut. It is a complicated environment, and from reviewing the domain and problem files it is hard to pick out reasonable agents, though multiple guesses can be made as to which parts of the domain could correspond to agents.

The decompositions returned separate out parts based on their surface-conditions along with their colours or treatments. Different saws in the domain are also separated as agents. However, these decompositions do not correspond to all the parts in the domains, and it is only a small set of them that can be modelled as agents. Comparing the problem files with the returned decompositions, it is hard to see exactly why the decompositions returned were as they are.

From the table, the obvious outlier is problem 10. This is a very simple problem version containing only three parts (others contain over thirty), and all the parts are separated as agents. This is the only case where all the parts can be separated and this is only because the domain is so small that it doesn't contain the interdependencies between variables that exist in the more complex versions of the domain. This domain is

solved in 0.00 seconds by ADP, but is also solved in 0.01 seconds by FF so this doesn't say anything about the usefulness of the decompositions. It will be interesting to see if the erratic structure found by the decompositions in the other problem instances is exploited by ADP.

5.2.4 Summary of Agent Decomposition Results

In summary, over the obviously multiagent IPC domains the algorithm performs as intended and the decompositions found generally correspond with those hand-crafted in the literature. In fact, it returns more detailed decompositions than might be expected, occasionally extending agents beyond the initial obvious decompositions. The results also exhibit the expected property that the algorithm does not return a decomposition for single-agent problem instances.

It is possible to modify the IPC domains to test the decomposition algorithm further. For example, the sokoban domain consists of a single 'player' actor that moves through a grid-world pushing crates around. This domain does not return any decompositions over any of the IPC problem instances. However, if other 'player' objects are artificially added to the problems, then the decomposition algorithm splits the players up as separate agents and returns the corresponding decomposition. Though, to do this, care must be taken that the additional 'player' object is relevant to the problem otherwise it is compiled away during FD's preprocessing.

Similarly, running the decomposition algorithm on the Bridges and Switches domains from Chapter 6 returns the expected agents. It should be noted that the algorithm has no trouble even with large problem instances that contain over three hundred thousand possible actions. In these cases, as with all other problems tested, the decomposition part of the algorithm takes less than 0.01 seconds.

The found decompositions can be used as input for ADP, to create a full, multiagent planning algorithm. For a planning algorithm that can solve any classical planning problem, as the decomposition algorithm can be performed as part of the preprocessing step, a standard single-agent planning algorithm can be used for the problems for which no decomposition can be found. This should lead to a planner that performs no worse on standard problems, but has increased performance on multiagent problems depending on the effectiveness of ADP, which is discussed in the next section.

5.3 Evaluation of ADP

The aim of the evaluation of ADP was to test the following hypothesis.

> **Hypothesis 2:** It is possible to design a multiagent planning algorithm that successfully exploits the multiagent structure of decomposed planning problems to plan faster than state-of-the-art single-agent planners.

To test this, ADP was run over every domain for which a decomposition was found. As ADP is implemented as an extension of the FD planner it is compared to other planning algorithms already implemented in that framework. The FD planner has had recent success with versions winning multiple tracks of the 2011 International Planning Competition. ADP was compared to the FF algorithm and LAMA planning algorithm. FF was chosen because it represents running ADP without using a decomposition. Because ADP utilises the same heuristics as FF, along with methods for applying these to multiple agents, this comparison shows directly how useful a multiagent approach is. LAMA was chosen because it is a recently successful planning algorithm that also uses the causal graphs as generated by FD. Instead of using them to create agent decompositions, LAMA uses them to create a hierarchical decomposition and find landmarks that can guide the planning process Both planners compared to are heuristic satisficing planners that return the first plan found as opposed to attempting to find the optimal plan for the domain.

The presentation of the results focuses on search time as this makes the most sense for a heuristic planner designed to return solutions as quickly as possible with no guarantees about plan quality. The search time presented for ADP includes the time taken to run the agent decomposition algorithm though this is dominated by the actual search time in all cases tested and never goes above 0.01 seconds. For comparison, during plan search in the largest problems, the variable set is iterated over hundreds of thousands of times as this is required to reset the relaxed planning graphs between heuristic calculations, whereas the decomposition algorithm will usually iterate over the variables set just twice. Results have been omitted for problems which have solution times less than 0.1 seconds by all three planning algorithms or that were not solved by any of the algorithms within the time limit. The totals displayed in the table are only for problems for which each planner returned a solution within the time limit and therefore may be less then the total for that respective column of the table.

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.01	0.1	0.19	90	97	**78**	**127**	540	470
0.01	0.11	0.28	**97**	128	128	**158**	725	1170
0.01	0.13	0.15	97	102	**92**	**150**	805	537
0.02	0.09	0.14	**71**	77	75	**112**	359	310
0.02	0.13	0.18	**59**	72	**59**	**114**	485	354
0.03	0.2	0.29	**104**	134	123	**194**	616	768
0.02	0.42	0.7	**123**	163	132	**203**	1520	1790
0.06	1.06	1.72	**156**	184	173	**287**	2760	2350
0.06	1.16	2.79	**210**	289	253	**483**	3600	4330
0.04	0.54	1.5	**141**	156	156	**236**	1500	2180
0.12	1.8	7.65	**331**	371	381	**628**	4460	7220
0.1	1.09	8.64	**228**	269	235	**606**	2590	6580
0.15	5.95	76.9	**336**	403	377	**672**	10800	23900
0.16	4.19	15.7	**263**	322	300	**471**	3750	5110
0.19	5.04	19.8	**329**	390	367	**861**	6760	7950
0.25	7.96	32.1	**327**	365	383	**949**	5450	8850
1.25	30	169	**2960**	3520	3310	**6250**	46700	73900

Table 5.9: Planning times, cost and the number of states evaluated for problems from the Rovers domain

5.3.1 Analysis by Domain

The first part of the evaluation will take an in-depth look at the domains whose decompositions were explored in detail. As with the analysis of the agent decomposition algorithm, the first domain to focus on will be Rovers, the domain most commonly used in the multiagent planning literature.

5.3.1.1 Rovers

It was noted from the decomposition that Rovers has a large number of internal actions, and should therefore be easy to solve with a multiagent approach. If nothing else, ADP should show an improvement over FF, its single-agent counterpart, for this domain. The results for the Rovers domain are shown in Table 5.9. The first set of columns show the search time in seconds, including the decomposition time for ADP. The second set of columns show the cost of the returned plans. While ADP is not optimised to return low

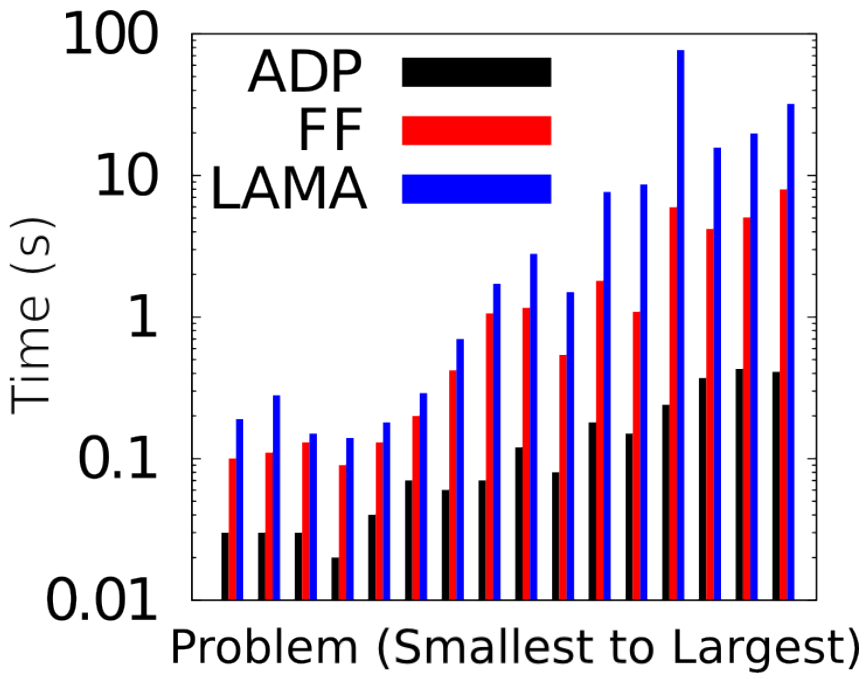

Figure 5.1: A graph of the search times for the Rovers domain with a log scale.

cost plans, this can be used as a reference point for the length of the plans found and the general quality of the returned plans. The final column shows the number of states evaluated by each algorithm, this shows how efficiently the planner navigated the search space. In each column, the best results between the three planners is marked in bold.

The table shows that ADP is significantly faster than both FF and LAMA over all the Rovers problem instances. The total search time for ADP is 2.33s compared to 30s for FF and 169s for Lama. This represents greater than 1,200% increase in planning speed by using a multiagent approach when compared to FF. It was expected that ADP would scale much better than the competing planners as the problem complexity increased. This was because the problem complexity was increased by adding more agents without significantly extending the environment they are operating in. Indeed, the planning times for ADP appear to be scaling much better with the increase in problem size than the other two planners. This is shown more clearly in Figure 5.1 which plots the search time for the rovers domain using a log scale for the y axis. On the most complex problems ADP is finding a solution in less than a quarter of a second while LAMA takes half a minute.

One interesting point is that at no point in the algorithm is a second round of relaxed planning graphs required. From the initial state, all goals are reachable by at least one

agent and an agent achieving a goal does not hinder the progress of other agents. For example, the following shows the estimated goal costs for each goal from problem 10 of the Rovers domain after the first round of planning graphs has been created (in the actual ADP algorithm not that the larger values in each column overwritten and do not need to be stored):

```
Agent0:  3  3  3 -- -- -- -- --  3 --  2
Agent1:  6  6 -- -- --  4  5  5 --  3 --
Agent2: -- -- --  3  2  2  3  3  3  3  3
Agent3:  5  5  5  6  5  5  3  4  5  6  4
```

This output can be read as Agent 0 had estimates of 3 for goal propositions 1,2,3 and 9, an estimate of 2 for goal proposition 11, and could not reach the other goal propositions. Meanwhile Agent 3 could reach all goals, but each costed more than at least one other agent. It can be seen that, while it is not possible for every agent to achieve every goal, each goal is achievable by at least one agent, and in this particular case by at least two agents.

The costs of the returned plans are lower for ADP than for the other two planners with ADP's collected plans containing 560 less actions than FF and 350 less than LAMA. This is roughly a 10-20% reduction in plan cost. This is interesting because ADP utilises the FF search algorithm, albeit on single-agent subproblems, for large parts of its search. In the final problem for the Rovers domain, the goal decomposition subprocess is called 10 times, which works out as once for each 32.7 actions added to the plan, meaning that the majority of the search takes place in the FF search on agent subproblems. This suggests that the route through the search space found by the goal decomposition is efficient at breaking down the problem.

Finally, the number of states evaluated by ADP is significantly lower than both FF and LAMA. ADP is evaluating 5,000 states per second compared to FF's 1557 and LAMA's 437. This can be explained by the low number of times that the goal decomposition process is called. The goal decomposition process is more costly than the heuristic calculations in both FF and LAMA but this is only called rarely in Rovers and the rest of the time ADP works on the reduced individual agent subproblems for which heuristic values can be calculated efficiently.

In summary, ADP manages to return accurate heuristic values that can efficiently navigate the search space and return low-cost plans whilst also managing to work almost exclusively with single-agent subproblems so that the plans are found significantly

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.02	0.04	0.18	76	74	**62**	**150**	357	899
0.01	0.02	0.14	**44**	50	46	**55**	137	942
0.02	0.06	0.18	71	61	**55**	**152**	527	1030
0.03	0.05	0.5	88	76	**68**	377	**356**	3580
0.02	0.12	0.13	111	101	**98**	**222**	1330	818
0.07	0.13	**0.04**	108	106	**75**	639	692	**175**
0.14	0.29	3.26	128	133	**92**	**936**	1070	9420
0.07	0.43	**0.05**	**100**	145	**100**	**102**	1230	105
0.15	0.19	0.25	**139**	140	**139**	**201**	207	274
0.55	1.51	**0.31**	185	186	**128**	1690	2010	**296**
1.14	2.56	3.02	207	189	**140**	3420	3120	**2520**
0.55	1.01	3.5	160	**136**	137	1260	**1010**	1920
0.48	19	16.9	**194**	**194**	**194**	**279**	6810	4630
1.02	16.2	3.23	235	228	**191**	622	3160	**607**
1.28	34.1	8.43	247	313	**229**	**513**	4630	845
2.74	43.3	9.34	284	347	**269**	1010	3740	**854**
1.11	30.4	85.1	**288**	327	290	**398**	4620	12900
18.7	111	–	534	**496**	–	10500	**8240**	–
2.43	–	20.6	**359**	–	361	**528**	–	1490
9.4	149	135	2670	2810	**2310**	**12000**	35000	41800

Table 5.10: Planning times, cost and the number of states evaluated for problems from the Satellites domain

faster. Not only does ADP perform better than competitive single-agent planners, but the improvements are significant in every area. This is a very strong result, but is currently only shown for a domain with a very well-behaved multiagent structure. The results for more complicated domains are discussed below.

5.3.1.2 Satellites

The next domain under analysis is the Satellites domain, the results for which are shown in Table 5.10. The analysis of the agent decompositions showed that the decomposed Satellites domain contains no influenced actions, which suggests that ADP should be especially effective. As in Rovers, ADP outperforms both FF and LAMA in terms of

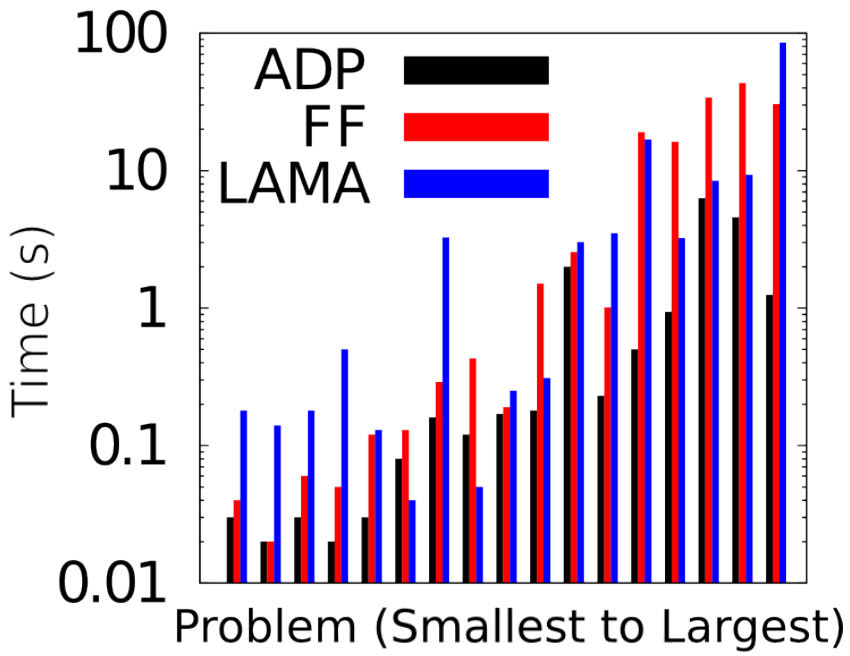

Figure 5.2: A graph of the search times for the Satellites domain with a log scale.

overall search time and the number of states evaluated. There are a few cases where LAMA finds a solution faster, by taking a significantly shorter route through the search-space, but the overall results show that ADP performs significantly better with a total time taken of 9.4s compared to the 135s of LAMA.

The results show that ADP is no longer returning the cheapest plans, with LAMA finding shorter plans in almost every case. As ADP still finds lower cost plans than FF, this results is likely due to the different structure that LAMA exploits being more applicable to the Satellites domain than to Rovers.

One observation from the decomposition results was that the Satellites domains tend to increase in environment complexity as they get harder, rather than just increasing the number of agents acting in the environment. It was speculated that this would mean that ADP scales better to the more complex problem instances on Rovers than it will on Satellites. Figure 5.2 shows the graph of the search times for comparison with the Rovers results. It is hard to draw anything conclusive from the graph; in fact, ADP is the only planner that manages to find a solution for all the planning problems within the time limit, and finds solutions in a couple of seconds that takes FF more than 5 minutes to solve. It appears that the previous speculations did not take into account the full nature of the problem. While the environment complexity does increase significantly, all actions in the domain are internal and there are no influenced actions so it is still

only each agent's subproblem that is made more complicated. Because of the lack of influenced actions, the coordination problem does not exist and is therefore not affected by the increase in the size of the environment.

In summary, ADP again outperforms the competing planners by a significant margin solving the combined problems solved by all planners within the time limit in 9.4 seconds compared to 149 and 135 for FF and LAMA respectively. Satellites is shown to be another example of a loosely coupled domain and one which a multiagent approach can perform well on, and ADP is shown to successfully exploit that multiagent structure by a convincing amount. The next domain under consideration contains a large number of both influenced and influencing actions and therefore may have a complicated coordination problem.

5.3.1.3 Logistics

The results of ADP on the Logistics domain are shown in Table 5.11. The results shown are for the problem set from 1998 as the problems in the 2000 problem set are all solved in less than 0.1 seconds by each of the planning algorithms tested. ADP was expected to show less of an improvement in this domain due to the large number of both influenced and influencing actions in this domain. However, again, ADP shows a significant performance increase over the other planning approaches. The total planning time for ADP is just 20.4 seconds while FF takes 388 and LAMA 191, ADP explores significantly less states than the other algorithms for this domain, with the number of states explored being not much more than the cost of the plan output, meaning that an almost direct route was taken through the search space.

A first thought would be that, while there are many possible interactions between the agents, these are not required to solve the problem, meaning that the coordination problem that actually needs to be solved is relatively trivial. However, analysing the output files shows that multiple rounds of relaxed planning graph generation are needed for all goals to be reachable. For example, for problem one, the output of the goal decomposition from the initial state is shown in Figure 5.3.

The figure shows that 3 rounds of relaxed planning graph generation is needed in order for all goals to be reachable. The fourth goal proposition is only completable by agent2 using at least one proposition added in the second round by a different agent and that proposition, in turn, must have required a proposition added in the first round by yet another agent so there is certainly coordination required. One thing that can be noticed from the figure is that agents 3, 4 and 5 do not appear to contribute to the problem.

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.07	0.16	0.26	102	**95**	97	**122**	667	875
0.08	0.26	0.34	117	116	**111**	**161**	1250	1100
0.12	0.16	0.35	83	80	**73**	**103**	432	530
0.13	0.2	0.25	102	**98**	99	**140**	688	666
0.54	2.53	2.14	184	191	**182**	**231**	3670	2250
0.23	2.14	0.61	**163**	175	**163**	**236**	4380	873
0.33	1.13	1.6	165	167	**153**	**212**	1740	1910
0.29	0.58	0.75	**116**	121	121	**139**	1050	1070
2.45	219	35	327	334	**318**	**456**	141000	14800
0.05	0.21	0.31	122	**121**	125	**201**	1300	1160
0.12	0.08	**0.07**	50	**46**	**46**	**53**	153	142
0.93	7.14	6.77	217	233	**206**	**263**	5670	3580
1.71	3.29	4.71	230	217	**208**	**536**	3150	3890
0.36	3	5.39	**164**	178	168	**236**	3630	2920
9.84	121	85.9	**313**	316	**313**	**453**	21500	10900
2.05	23.9	40.4	364	371	**357**	**516**	16900	15600
1.14	2.8	5.87	**158**	159	159	**196**	2050	2570
20.4	388	191	2980	3020	**2900**	**4250**	209000	64800

Table 5.11: Planning times, cost and the number of states evaluated for problems from the Logistics domain

```
Agent0: -- -- -- --  0 --
Agent1: -- -- -- --  0 --
Agent2: -- -- -- --  0 --
Agent3: -- -- -- --  0 --
Agent4: -- -- -- --  0 --
Agent5: -- -- -- --  0 --
Agent6:  4 -- -- --  0  3
Agent7:  4 -- -- --  0  3
--next layer--
Agent0: -- -- -- --  0 --
Agent1: -- -- -- --  0 --
Agent2: 12 -- -- --  0 11
Agent3: -- -- -- --  0 --
Agent4: -- -- -- --  0 --
Agent5: -- -- -- --  0 --
Agent6:  4 11 11 --  0  3
Agent7:  4 11 11 --  0  3
--next layer--
Agent0: -- -- 19 --  0 --
Agent1: -- 19 -- --  0 --
Agent2: 12 -- -- 18  0 11
Agent3: -- -- -- --  0 --
Agent4: -- -- -- --  0 --
Agent5: -- -- -- --  0 --
Agent6:  4 11 11 --  0  3
Agent7:  4 11 11 --  0  3
```

Figure 5.3: The output of the relaxed plan generation algorithm from the initial state of Logistics problem 1

However, it may be that they are needed to contribute towards some of the generated subgoals as this is not shown in the simple output of goal costs for each agent.

The last thing to note about this domain is that it has a larger number of agents than any of the others. This domain presents the best test of ADP's ability to perform as the number of agents increases and ADP is shown to be significantly faster in problems with over a hundred agents. In summary, this is an important result, as it shows that ADP can provide significant performance increases even in domains which require coordination between the agents. The goal decomposition algorithm must be finding the correct subgoals and distributing the goals amongst the agents correctly.

5.3.1.4 Airport

Airport is another domain for which multiple rounds of relaxed planning graphs are required. The results for this domain are shown in Table 5.12. The Airport domain is interesting as it contains a large number of problem instances increasing in complexity beyond the other domains to the point where there are many problems that both FF and LAMA can not solve within the time limit, which allows for the capabilities of ADP to be tested in much more complicated domains. As these domains are not solvable by all planners, it is obvious that an efficient route through the search space is needed as it is not possible to completely explore it within the time limit.

As may have come to be expected, ADP shows significant reduction in planning time compared to both FF and LAMA. However, there are a few domains for which ADP does not return a result but both FF and LAMA do. Analysing the output files shows what is happening here. The individual goal search terminates in a dead end of the search space due to an action added at the beginning of the agent planning process. It takes a long time for the search to backtrack to the point where it finds a working goal decomposition as it ends up exploring all the states in between. This is because the airplanes can get in the way of one another, possibly blocking off another plane from ever reaching its goal. In FF and LAMA, this sort of blocking is noticed immediately and the search backtracks, in ADP, backtracking induced by global dead ends can only occur once goal decomposition is called again.

Of course, ADP is a heuristic algorithm and, as such, cannot perform well in every case. However, the few times where it ends up following a bad direction, it is not able to recover and does not solve the problem within the time frame. Having said this, there are three problems for which ADP does not return a solution while there are thirteen for which only ADP returns a solution so ADP is clearly the best performing planner over

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.01	0.27	0.02	**60**	64	**60**	**65**	4740	79
0.02	2.91	0.15	**79**	81	83	**97**	29600	816
0.03	6.84	1.08	**111**	**111**	**111**	**138**	47400	3590
0.04	3.72	1.45	94	92	**90**	**102**	32700	6010
–	26.4	**3.31**	–	**121**	**121**	–	145000	**9050**
0.04	0.11	0.07	**101**	**101**	**101**	**104**	493	139
0.1	15	**0.1**	**148**	**148**	**148**	**160**	40000	161
0.14	33.9	0.15	**168**	**168**	**168**	183	64800	**175**
0.14	44.3	1.63	166	164	**162**	**177**	77000	1690
0.29	63.7	–	**212**	**212**	–	**304**	83500	–
0.29	–	–	**220**	–	–	**256**	–	–
0.29	–	–	**252**	–	–	**267**	–	–
0.59	–	–	**296**	–	–	**468**	–	–
0.75	–	–	**310**	–	–	**410**	–	–
0.51	–	–	**312**	–	–	**326**	–	–
1.13	–	–	**360**	–	–	**650**	–	–
0.93	–	–	**392**	–	–	**411**	–	–
1.62	–	–	**397**	–	–	**757**	–	–
3.15	–	–	**437**	–	–	**565**	–	–
0.12	**0.09**	0.11	111	111	**109**	122	124	**118**
0.25	40.5	63.2	157	220	**155**	**179**	39900	35200
–	**7.82**	–	–	**195**	–	–	**6790**	–
–	**53.4**	–	–	**181**	–	–	**67500**	–
0.72	–	23.4	**283**	–	315	**530**	–	10200
0.38	–	–	**229**	–	–	**273**	–	–
0.46	–	–	**251**	–	–	**290**	–	–
2.53	–	–	**423**	–	–	**537**	–	–
2.49	–	–	**483**	–	–	**654**	–	–
0.89	148	68	1200	1260	**1190**	**1330**	337000	48000

Table 5.12: Planning times, cost and the number of states evaluated for problems from the Airport domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.11	**0.1**	0.31	55	41	**36**	**950**	2010	3300
0.48	0.74	**0.26**	71	61	**55**	4030	10600	**2200**
2.39	17.7	2.76	105	194	**87**	**12600**	166000	15700
–	–	**18.9**	–	–	**143**	–	–	**54100**
1.38	5.23	1.74	**38**	61	42	**5500**	42100	9210
5.52	78.6	8.71	108	152	**94**	**12500**	275000	22400
1.05	**0.4**	1.2	35	**30**	31	6800	**4640**	8280
10.8	12.8	**5.41**	130	92	**70**	22300	50000	**14900**
17.9	–	**17.7**	115	–	**95**	**18400**	–	25500
4.62	–	**0.4**	51	–	**32**	9320	–	**1080**
49.7	92.3	**12.8**	161	134	**105**	51200	148000	**16900**
0.29	0.07	**0.02**	**28**	32	**28**	991	373	**96**
0.15	0.1	**0.04**	28	29	**25**	**66**	124	95
19.1	9.86	**0.32**	110	72	**59**	11600	12400	**325**
3.96	12.6	5.82	61	54	**45**	**8520**	44100	14900
0.38	0.49	**0.2**	36	**33**	34	**102**	135	162
59.9	**54.2**	131	**113**	119	129	**5490**	11200	21600
155	285	171	1080	1100	**840**	142000	767000	**130000**

Table 5.13: Planning times, cost and the number of states evaluated for problems from the Depot domain

these domains under any sensible metric.

5.3.1.5 Depot

The depot domain was interesting as it contained almost exclusively public actions. This means that the agent decomposition is having little effect and it was expected that ADP would perform roughly evenly with FF. The results are shown in Table 5.13 and, while ADP is still the fastest performing planner overall, the difference is much smaller than in the other problem domains. In fact, ADP is only the fastest planner in 3 of the problem instances and, LAMA is faster most of the time.

That ADP is showing a slight improvement in planning times even with over 90% public actions is surprising. This has to be due to the quality of the heuristic values it finds and the success of the goal decomposition process because the individual agent

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
3.96	5.09	13.6	**1360**	1750	1720	**9670**	21700	21600
3.44	**3.22**	15.3	**1530**	1700	1930	**8900**	12200	22300
6.08	**4.74**	18.4	**1370**	1400	1770	**15200**	16100	25300
–	–	**0.23**	–	–	**1500**	–	–	**191**
–	–	**0.29**	–	–	**2160**	–	–	**265**
10.2	**8.08**	29.5	**1760**	2130	2070	**20700**	22200	30200
5.47	–	11.7	**1610**	–	1830	**10500**	–	15000
9.05	**7.46**	18.3	**1580**	1710	1680	**16100**	24500	24000
4.88	**3.66**	6.59	**1280**	**1280**	1370	13500	13500	**9260**
8.9	11	37.5	**1900**	2200	2460	**16600**	28200	38100
15.3	**12.2**	38.8	**1720**	2000	2270	**22200**	26700	32100
16.3	–	60	**1930**	–	2550	**28800**	–	57900
6.71	–	**0.24**	**1630**	–	1940	12200	–	**225**
22.2	–	80.7	**2390**	–	3110	**32100**	–	62000
11.3	–	**7.49**	**2020**	–	2420	18900	–	**6160**
2.91	2.32	**0.21**	**1120**	1210	1230	7480	7890	**139**
9.49	**4.34**	16	**1560**	1670	1590	**13600**	15200	14900
4.12	–	9.49	**1270**	–	1350	10100	–	**6920**
6.61	**5.74**	46.9	**1630**	1810	1960	**12700**	14500	25200
80.8	**67.9**	241	**16800**	18800	20000	**157000**	203000	243000

Table 5.14: Planning times, cost and the number of states evaluated for problems from the Woodworking domain

subproblems are barely smaller than for the full problem. Indeed, the table confirms this because the number of states evaluated is significantly lower than FF. As LAMA can be thought of as a non-multiagent way of exploiting domain structure, it makes sense that it performs better than ADP on this problem where the multiagent structure is minimal.

5.3.1.6 Woodworking

The final domain that will be looked at specifically is Woodworking, for which the returned decompositions were not as intuitive as for most of the other domains. This domain has a large number of public actions, a large number of the internal actions are bot influenced and influencing and it has an unintuitive decomposition. The results show

that FF's total time, over the problems for which all three algorithms returned a solution, is faster than ADP's. However, looking more closely, there are six problem instances for which ADP returned a solution within the time limit while FF did not. ADP also continues to have the lowest number of states evaluated of the three algorithms.

This domain shows that, while there may be situations where ADP is not the best choice, it is still competitive over domains for which the decompositions found induce complex multiagent problems.

5.3.2 Collated Results

Table 5.15 shows the collated results for each of the domains for which a decomposition was found. The detailed results for those not discussed in the previous section are included in the appendix.

The results show a massive increase in performance of ADP over FF taken as a whole over all the domains tested. This performance is larger than the number of agents involved in the decompositions showing that this approach is a significant improvement to current state-of-the-art planning algorithms over domains with a multiagent structure. As the agent decomposition algorithm is negligible in terms of time taken it can be integrated into an extra preprocessing step that can then decide whether or not to invoke ADP or a single agent planner on the given problem. As around a third of IPC domains have an agent decomposition and ADP makes significant improvements in average planning times for these domains this would result in a very effective heuristic planning algorithm. Overall, the results certainly corroborate the hypothesis, showing that state-of-the-art planners can be significantly outperformed by a multiagent approach.

5.3.3 Discussion: Design Decisions

Many different versions of the ADP were created and tested on the road to the version presented in this thesis, which proved to be the most robust whilst also allowing for the largest improvement on the domains that it was successful in. This section discusses the different versions and design decisions made and discusses their impact on the overall planning performance.

5.3.3.1 Relaxed Planning Graph Generation

Perhaps the most intuitive method, and the first to be considered when applying relaxed planning graphs to a multiagent problem is to transfer messages between agents every

Domain	Search Time (s)			Plan Cost			States Evaluated		
Name	ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
Airport	**0.89**	148	68	1200	1260	**1190**	**1330**	337000	48000
Depot	**155**	285	171	1080	1100	**840**	142000	767000	**130000**
Driverlog	**27.4**	68.9	30.5	**867**	1150	1030	189000	451000	**125000**
Elevators	**0.02**	0.36	0.25	389	471	**373**	**130**	2150	1040
Floortile	**3.61**	57	297	524	335	**317**	**72700**	3900000	7980000
Logistics(98)	**20.4**	388	191	2980	3020	**2900**	**4250**	209000	64800
Pathways	112	204	**9.46**	3640	3390	**3370**	487000	1.08E7	**46000**
Rovers	**1.25**	30	169	**2960**	3520	3310	**6250**	46700	73900
Satellites	**9.4**	149	135	2670	2810	**2310**	**12000**	35000	41800
Storage	15.1	**1.52**	4.22	442	**152**	215	185000	**23500**	35200
Tpp	**0.27**	36	94.2	**518**	2620	2520	**531**	23500	36000
Transport	**0.95**	100	41.1	14100	20100	**13700**	**2630**	42000	13200
Woodworking	80.8	**67.9**	241	**16800**	18800	20000	**157000**	203000	243000
Zenotravel	**0.5**	2.91	4.99	**448**	472	490	**1270**	6860	4880

Table 5.15: Total planning times, cost and the number of states evaluated for all problem domains.

time an influencing action is added to a graph. This way, each agent creates their own graph, while all reachable propositions will eventually be found. This also allows for in depth bookmaking about which agent is passed which proposition and at which time.

However, a version of this approach was implemented and shown to be strictly worse than single-agent FF. Essentially all that this approach achieves is creating the full relaxed planning graph, but in a more convoluted way that involves extra computation. While it is possible to extract more information from the graphs because of the multiagent decompositions, essentially, corresponding to the heuristic calculation used in ADP, the price of creating the graphs is too high. There are too many messages that need to be sent between the agents. A heuristic needs to be easy to calculate and generally point the search in the right direction and this approach fails on the first part. ADP improves on this because the messages passing effectively only occurs once per round of planning graphs which is a huge speedup in heuristic calculation.

5.3.3.2 Goal Decomposition

The goal decomposition method in ADP selects the agent that has the most independently achievable goals to be the next one used in the individual agent search part of the algorithm. Other methods tested included select the first goal found or the agent with the lowest cost goals, or to solve subgoals, or head for goals in the lowest layer first. For each possibility, there were some problems that were solved faster and some that were solved much slower. The method used in ADP had the best coverage overall with the most problems solved within the time limit. While it is certainly possible to solve some of the problems even faster, as ADP was already outperforming the other planners to a large extent, it seemed better to choose the option that led to the most robust planner.

5.3.3.3 Early Dead-End Verification

While an agent is working on its individual subproblem, it is possible to add an action that inadvertently prevents another agent from ever completing their goals. In ADP this is not checked until the agent finishes its goal set and the goal decomposition process is called again. While this clearly does not hinder ADP much in the majority of problems, there were many domains for which a large proportion of search time was spent after already reaching a dead end in the global search space.

Early Dead-End Verification performs relaxed planning graph generation during planning for individual agent subproblems in order to make sure that the state is not

blocking off other agents. This only needs to be performed when an influencing action is added to the search, but for the vast majority of problem instances the cost of performing this extra check outweighed the potential benefits. However, this is possibly an interesting direction for future work and an area that ADP could be improved further.

5.4 Summary

This chapter presented the evaluation of the agent decomposition algorithm and ADP in order to see if they corroborate the hypotheses from the introduction. The decomposition algorithm was shown to return the expected decomposition for all the domains commonly cited as multiagent. It also found sensible decompositions in many other domains and only returned a few that were not easily identifiable. ADP was shown to significantly outperform current state-of-the-art planners, even for domains which require large amounts of coordination between the agents. The results of these experiments show that ADP is successful at exploiting multiagent structure and also that the agent decomposition algorithm is successful at finding structures with good properties for planning.

Chapter 6

Possible Extensions

The previous chapters introduced the core contributions of this thesis related to multiagent classical planning problems that are as close to the well-defined single-agent classical planning paradigm as possible. However, this means that there are many important multiagent concepts that have not been covered. The two of these most suited to planning are dealing with self-interested agents, that have their own goals and preferences over the plans that achieve them, and concurrent actions, that model situations where agents must coordinate simultaneously, such as when lifting a heavy object together. The first part of this chapter is concerned with the former, while the latter is explored in the second part of this chapter. In line with the overall methodology of the thesis, these multiagent concepts are introduced as extensions to the multiagent classical planning formalism introduced in Chapter 3.

The extensions discussed in this chapter introduce extra complexities to multiagent classical planning. Planning under strategic considerations is much harder than in the completely cooperative case. It is no longer sufficient to simply find a plan that achieves all goals. Instead, plans must have certain properties, which can depend on all other possible plans, that ensure that they are rational choices. Planning with concurrent actions is especially difficult as it embraces the interactions between agents and requires using joint actions. Once strategic concerns or concurrent actions are taken into account it is no longer possible to provide generalised planning methods that work efficiently over large numbers of domains. Instead, the aim of this chapter is to provide a basis for future work and to improve on results already in the literature.

6.1 Multiagent Classical Planning with Self-Interested Agents

It has already been observed that modelling self-interested agents is natural in a multiagent planning environment. It was shown, even under the assumption that agents are completely cooperative, that agents having autonomy over part of the domain is a key concept in multiagent planning (Section 3.1). In multiagent systems, this autonomy normally brings with it the ability for agents to decide their own goals and the best way to achieve them, which, in turn, suggests that the agents are self-interested, concerned with themselves over the system as a whole.

As the subject matter moves further from the single-agent classical planning starting point, it is naturally more difficult to provide definitive answers and results. Working in a self-interested environment is more complicated than solving problems in which all agents are assumed to work together for the common good. In fact, not only is it harder to solve problems, it is no longer even clear what a solution to a problem consists of. It may not be possible to achieve all agents' goals with a single combined plan, and improving the planning cost for one agent may hurt the planning cost of another.

This chapter extends the Coalition Planning Games (CoPGs) framework, introduced in Section 2.5.1, modified to be applicable to MMPTs. The solution concept used in the literature was found to have unintuitive consequences in certain planning domains, so a new solution concept is defined. This is based on the observation that farsightedness, the ability to look beyond the short-term consequences of actions and deviations, is inherent to the nature of planning. Therefore, it is important to include some notion of 'farsightedness' within the solution concept used for multiagent planning domains.

To verify that a solution is stable under the definition presented in this chapter requires finding all possible plans for a number of planning problems exponential in the number of agents in the domain. Even so, a subclass of planning problems is found for which it is possible to find stable solutions and an algorithm is introduced that can find such solutions using existing planning technology.

6.1.1 Strategic Multiagent Planning

In strategic multiagent planning it is assumed that agents have individual goals and preferences over the plans that achieve them, breaking the simple goals assumption of classical planning. The preferences over plans are provided by a utility function

that measures the 'worth' of a set of plans for each agent. It is assumed that each agent is rational, and the rationality of all other agents is common knowledge. These assumptions allow for reasoning about other agents' behaviour, and therefore for making predictions about how the rest of a system will behave, which brings some semblance of order to an otherwise erratic multiagent problem.

The formal definition of a strategic MMPT is relatively straightforward and follows the extension to MA-STRIPS introduced in Section 2.3.1. The only difference for MMPTs is that now the goal set is distributed amongst the agents. For this presentation, it will be assumed that all actions are performed asynchronously, though it is possible to apply the definition to the concurrent action specification introduced in the latter half of this chapter.

Definition A *Strategic Multiagent MPT (strategic-MMPT)* is a tuple

$$\Pi^* = \langle V, \Phi, I, G', X, A \rangle$$

where $\Pi = \langle V, \Phi, I, G, X, A \rangle$ is an MMPT with $G = \bigcup G'$ and $G' = \{G_1, \ldots, G_n\}$ contains a goal set for each agent $\phi_i \in \Phi$ such that each $g \in G_i$ belongs to the agent subproblem of ϕ_i.

The only new element of this definition is G', which is used to show that the goal set has been split up into one for each agent. Each goal in the associated MMPT is given to at least one agent in the strategic problem and each goal that belongs to an agent must appear in that agent's subproblem. In other words, an agent's goal must not include variables from the internal state of another agent. It is also assumed that the MMPT contains no public actions so that each action in the problem belongs to only one agent.

6.1.2 Assumptions and Equilibrium Concepts

The solution to the multiagent planning problems considered in the rest of this thesis is simply a plan that leads from the initial state to the goal state. In the strategic case, the presence of agent preferences means that certain solutions to the completely cooperative problem no longer count as solutions. One task of game theory is to work out which solutions will form based on the rationality assumptions about the agents. To do this, equilibrium concepts are defined that model solutions where no rational agent would ever want to change to a different solution. If no-one would want to change, then the solution is stable and can be expected to be maintained and agreed upon by all the agents in the system.

> An equilibrium is a *joint* solution for all the agents, such that there is no reason for any agent to change their own choice of actions given their desire to maximize some real-valued utility function. (Bowling et al., 2002)

However, there are many possible equilibrium concepts depending on the exact nature of the assumptions about the agents' rationality and capabilities. For example: Can agents agree to help each other? Can they offer exchanges of utility in return for certain actions being performed? How does another agent's deviation effect the execution of plans? These assumptions will generally be tailored to the particular problem being modelled and it is impossible to give comprehensive coverage of all the possibilities. However, there are certain assumptions that are closely linked with the general nature of multiagent planning, and these will be the assumptions considered in this thesis.

The first assumption comes from noticing that planning problems such as the one shown in Figure 6.2 (right) on page 148 requires cooperation between the agents if any one of them wants to achieve their goal. This type of feature is prevalent in planning in general as it appears in any problem that involves non-concurrent coordination. This represents one of the two types of possible non-concurrent interactions (introduced in the next section) and occurs any time an agent's action changes an environment variable to one that is needed by another agent. It is therefore likely to appear in almost any multiagent planning problem.

Standard, non-cooperative game theory, does not contain mechanisms for dealing with such coordination. Instead, it models agents that are not willing to help each other but are only working for themselves. However, cooperative game theory, specifically that which assumes that coalitions of agents can form, allows for solutions to problems containing non-concurrent coordination. It therefore makes sense in multiagent planning to assume that coalitions can form.

Coalition Planning Games (Brafman et al., 2009) were introduced in Section 2.5.1. They take the idea of coalitions from cooperative game theory and apply them to multiagent planning problems. However, it was argued that the solution concept presented for CoPGs has some unintuitive consequences. As was shown in Figure 2.5 it allows for stable plans that contain inefficient action sequences, even when these action sequences do not interfere with any of the other agents in the domain. This violates the following property that it is intuitive to assume that rational agents would have:

> In a stable solution an agent will always perform its lowest cost plan (therefore maximising its utility) in areas where this has no impact on other agents in the domain.

There was a secondary problem with the solution concept in the literature, which was that it did not define the expected solution to the problem shown in Figure 2.6 as stable. This is due to not taking into account interference effects between critical actions. In this particular problem, there are no stable solutions as the way the deviations are handled creates an infinite cycle. The solution where B crosses the bridge to deliver its parcel is dominated by the coalition of A and B where A does all the work and in return B does not destroy the bridge. However, the solution where A and B are in a coalition is dominated by A just delivering its own parcel, as the fact that B would cross the bridge first is not considered by the deviation. The solution where A delivers its own parcel is clearly dominated by B delivering its own parcel which returns to the original solution idea.

The cycle between the potential solutions to Figre 2.6 is brought about because A's deviation from the coalition does not consider how this affects B's plan, in other words, agents do not look beyond the *immediate* consequences of their deviations. This leads to the second property of planning domains that suggests certain game-theoretic assumptions. Namely, that planning is focussed on look-ahead, the ability of an agent to reason about the future, so it seems reasonable to assume that agents can reason about the consequences of their own potential deviations.

In situations where the coalition involves performing particular helpful actions for the other agent, then assuming non–deviating agents do nothing is fine because this stops them from performing the helpful actions that they are only performing because of the coalition. However, when the coalition involves refraining from performing otherwise beneficial destructive actions, then assuming non–deviating agents do nothing does not take into account the destructive actions (such as crossing the bridge) that they would want to perform if not in the coalition. This is the problem of farsightedness, the idea that agents must look beyond the immediate consequences of their actions or deviations.

The problem of farsightedness is discussed in Ray (2007), 'A Game-Theoretic Perspective on Coalition Formation'. While this book has no direct links to planning, the arguments presented fit exactly with the problems with applying game-theoretic concepts to planning that were previously discussed. A main argument of Ray is that Von Neumann and Morgenstern's seminal work (Morgenstern and Von Neumann, 1944) has shaped the way that game theory research has developed, to the extent that problems with farsightedness are neglected. While general games are discussed, the majority of Von Neumann and Morgenstern's work is focussed on zero-sum games, games in which

a gain for one side is mirrored by an equal loss by the other. In these type of settings it makes perfect sense to consider that the other agent is out to disrupt you as much as possible. This is because every time they cause you to have a loss they achieve an equal gain. This idea is prevalent in the use of characteristic functions in cooperative game theory and means that considerations about the non-immediate effects of deviations are often ignored.

The prevalence of characteristic functions is also misaligned with the goals of planning itself. Characteristic functions define the worth of each possible coalition by analysing their potential contributions to the overall problem. There are many different ways of doing this but the overall idea is the same. The different methods operate under a variety of assumptions under which the worth of a coalition is calculated. For example, the α-*characteristic function* is based on the payoff that a particular coalition can guarantee for itself in the worst case scenario. As this has a simple definition, it is often used, and it is tempting to apply it to planning, but, as has been discussed before, the prevalence of interference actions means that it is almost always possible to block an opponent's plan, meaning that worst-case scenarios almost always involve a plan being completely undermined.

The point of the preceding discussion was to argue both, that a solution concept for multiagent planning must take into account farsightedness (as this is fundamental to planning itself) and, that the amount of research into solution concepts that can be directly applied to planning environments is limited. Most existing succinct representations of games assume some form of locality and/or compactness of the dependence of agents payoffs on the choices of other agents. This, however, almost 'by definition' requires the planning part of the system to be simplistic as the complexity of planning stems from the, possibly indirect, global dependencies between the agents' actions.

Instead of defining a solution concept directly for multiagent planning, another approach is to equate planning problems to extensive-form games, such as the one adopted in Larbi et al. (2007) discussed in Section 2.5.2.1. The problem with this approach, and related approaches, is that the translation method is incredibly inefficient, leading to a game that is much larger than the original planning problem. In general, planning methods can be seen as attempts to exploit the structure in planning problems in order to efficiently navigate large search spaces. By translating the planning problem to a game, this aspect is lost and it no longer becomes a planning question.

6.1.3 A New Solution Concept

The preceding discussion suggests the need for a new solution concept that can be applied directly to multiagent planning problems and that takes into account farsightedness, the ability for agents to reason about the long-term effects of deviations. The assumptions for this solution concept include that agents can form coalitions at no cost, and that these take the form of binding agreements about plans that are assumed to be automatically enforced once agreed to.

It is also assumed that all actions are performed sequentially with interleaved execution between agent's individual plans. So, for example, in a problem with three agents ϕ_1, ϕ_2 and ϕ_3, a complete solution will consist of equal length plans from each agent. As with concurrent actions, it is assumed that each agent has access to a 'noop' action that can be added to ensure that all plans are the same length. The plans of ϕ_1, ϕ_2 and ϕ_3 are combined into a joint plan by adding the first action of ϕ_1's plan followed by the first action of ϕ_2's plan and the first action of ϕ_3's plan and then the second action of each plan in order and so on until the full plan is formed.

One further assumption about the domains under consideration is that goals in them, once completed cannot be undone. This is called goal persistence. A problem has goal persistence if no goal propositions appear in the negative effects of an action. It is possible to ensure that all domains have the goal persistence property by adding a `claim-goal` action that has the original goal that needs to be modified as preconditions and a new `goal-claimed` proposition as its effect. This `goal-claimed` proposition then becomes the new goal to replace the old one. The reason that most of the domains presented in this paper include a 'reported' action is to ensure that they have goal persistence.

It has already been shown that it is not enough to simply define a stable solution as one where no subset of agents can deviate to increase their utility. To solve this problem it needs to be assumed that agents not in the deviating subset, instead of doing nothing, perform the best plan they can in the smaller subproblem created by considering the plan to be deviated to. This leads to a recursive definition of the solution concept.

Let $u_S(\pi)$ represent the vector of utilities for agents in $S \subseteq \Phi$ calculated from executing the plan π where the utility to agent ϕ_i is calculated as $N - cost(\pi_i)$ where N is some large number and $cost(\pi_i)$ is the cost of all the actions in ϕ_i's plan. Let $(\pi_S, \pi_{S'})$ represent the joint plan constructed by combining the plans π_S and $\pi_{S'}$ for disjoint subsets $S, S' \subseteq \Phi$. Call a vector $u > u'$ if every element of u is greater than the

equivalent element in u' and similarly $u \geq u'$ if every element of u is greater than or equal to it's corresponding element in u'.

With this, it is possible to define a solution π as *stable* if there doesn't exist a strategy π_S for any subset of agents $S \subseteq \Phi$, $S \neq \emptyset$ such that

$$u_S(\pi_S, \pi^*_{\Phi \setminus S}) \geq u_S(\pi)$$

and

$$\exists \phi_i \in S : u_i(\pi_S, \pi^*_{\Phi \setminus S}) > u_i(\pi)$$

where $\pi^*_{\Phi \setminus S}$ is the stable solution to the smaller planning problem over the set of agents $\Phi \setminus S$ formed by fixing S's strategy to π_S. If $(\pi_S, \pi^*_{\Phi \setminus S})$ is not a valid plan then it is assumed that $u_S(\pi_S, \pi^*_{\Phi \setminus S}) = 0$.

In other words, a solution is stable if there is no subset of agents that can deviate to a plan in which no agent is worse off and at least one agent is strictly better off and, instead of assuming non-deviating agents do nothing, it is assumed that they respond with the stable solution to the reduced planning problem given by setting π_S. Note that this reduced planning problem is strictly smaller than the previous problem so eventually the non-deviating set will be reduced to $\emptyset$ at which point the definition is grounded.

Consider how this applies to the problems shown in Figures 2.4, 2.5 and 2.6. In the former, the cooperative solution is stable, it is clear that no agent can ever achieve a better utility so $\nexists \phi_i \in \Phi : u_i(\pi_S, \pi^*_{\Phi \setminus S}) > u_i(\pi)$. The inefficient solution shown in Figure 2.5 is no longer stable because the deviation of the coalition of A and B to the improved solution is allowed under the solution concept. In the bridge example, if agent A is delivering both parcels, then the promising looking deviation to the plan where A only picks up its own parcel is not allowed because agent B's best response to this stops it from working (B crosses the bridge and then a gets no reward as it cannot achieve its goal).

6.1.4 Safe-MMPTs

On the face of it, it seems that it will be impossible to find stable solutions to strategic-MMPTs in any reasonable amount of time. The recursive nature of the definition means that it is hard to even check that a solution is stable, let alone to find one. Indeed, there probably is no practical general solution for finding stable solutions, however, it may be possible to find solutions for certain classes of MMPTs.

To this end, it is possible to define Safe-MMPTs, so called because in them, it is impossible for an agent's situation to be made worse by other agents that are only improving on their current plans.

Definition A safe-MMPT is an MMPT such that for all possible plans π and $\forall S \subseteq \Phi$

$$u_S(\pi') \geq u_S(\pi) \rightarrow u_{\Phi \setminus S}(\pi'_S, \pi'^*_{\Phi \setminus S}) \geq u_{\Phi \setminus S}(\pi_S, \pi^*_{\Phi \setminus S})$$

In other words, given a plan and a subset $S \subseteq \Phi$, if S deviates to a strictly better plan, then the other agents are not adversely affected by this deviation because their best response to the deviation is at least as good as their best response to the original plan. In these environments, an action that is detrimental to another agent's plan cannot be beneficial. It is easy to see that the parcel domain from Section 1.2.2 is safe, since it is never beneficial to pickup another agent's parcel (the only way to potentially hinder their plan) unless planning to cooperate with that agent. In fact, the switches domain, and any domain that contains coordination effects but no interference effects is safe. It is possible for a domain that contains interference effects to be safe, but they must only occur in a fairly restricted manner not relevant to the ability of agents to achieve their goals.

Clearly, safe-MMPTs have certain useful properties as there would otherwise have been no reason to define them. The reason is shown by the following proposition:

Proposition 6.1.1 *For a safe-CoPG* Π*, any plan* π *that satisfies the following is stable:*

- π *is a minimum total cost plan.*
- *For all agents* ϕ_i *in* Φ*,* π*'s cost to* ϕ_i *is less than or equal to the minimum cost for* ϕ_i *to solve its agent subproblem.*

Proof Call the output plan π^+. Assume by way of contradiction:

$$\exists \pi_S : S \subseteq \Phi S \neq \emptyset : u_S(\pi_S, \pi^*_{\Phi \setminus S}) \geq u_S(\pi^+)$$

and that

$$\exists \phi_i \in S : u_i(\pi_S, \pi^*_{\Phi \setminus S}) > u_i(\pi^+).$$

So

$$\sum_{\phi_i \in S} u_i(\pi_S, \pi^*_{\Phi \setminus S}) > \sum_{\phi_i \in S} u_i(\pi^+)$$

but from the definition of a safe-CoPG we know that

$$\sum_{\phi_i \in \Phi \setminus S} u_i(\pi_S, \pi^*_{\Phi \setminus S}) \geq \sum_{\phi_i \in \Phi \setminus S} u_i(\pi^+)$$

which also implies that $(\pi_S, \pi^*_{\Phi \setminus S})$ belongs to the set of all possible plans that achieve the goals. Putting this together gives

$$\sum_{\phi_i \in \Phi} u_i(\pi_S, \pi^*_{\Phi \setminus S}) > \sum_{\phi_i \in \Phi} u_i(\pi^+)$$

a contradiction with the assumption that the returned plan has minimum cost. □

6.1.5 Solving Safe-MMPTs

The result from the previous section suggests an obvious algorithm for finding solutions to safe-MMPTs. A metric-planner can be used and action costs and minimisation metrics modified to ensure that the conditions in the proposition are met. A metric-planner is a planner that includes the ability to deal with action costs and other integer numeric fluents and to minimise or maximise linear combinations of such fluents.

As a first step, an optimal planner is used to solve each agent's subproblem to get an upper bound on their individual contributions. Let $c(\phi_i)$ represent the cost of the optimal plan for ϕ_i's subproblem; then, the actions that belong to ϕ_i can be annotated so that they increase the cost of a function $\phi_i - cost$ that represents the cost of the plan so far to agent ϕ_i. Note that this process relies on the assumption that the strategic-MMPT contains no public actions so that each action only belongs to one agent. After this, the only change that needs to be made is that a metric is added that ensures that $\phi_i - cost \leq c(\phi_i)$ and that the total cost of actions is minimised in the returned plan. Using this algorithm with an optimal metric planner clearly satisfies the conditions in Proposition 6.1.1 and therefore returns a stable solution for safe-MMPTs.

Unfortunately, it turns out that the problems created by the algorithm are not easy for metric planners to solve. Metric-FF, a well-known metric planner that has shown good performance does not work well on these problems. Metric-FF follows a similar search process to that presented for the standard FF algorithm in Section 2.1.4.1 and is reliant on its EHC search strategy succeeding in order to return plans at high speed. Unfortunately, due to the large number of constraints created by the solution method, EHC fails in a large number of cases, especially in problems that involve cooperation between agents. Intuitively this is because the additional constraints that provide threshold values for each agent add a large number of dead-ends to the search space.

6.2 Concurrent Actions

The rest of this chapter looks at the extension to include concurrent actions. Concurrent actions model the ability for agents to interact simultaneously, therefore breaking the sequential actions assumption of classical planning. An example of a concurrent action would be multiple agents combining to lift a heavy object together. This models concurrent coordination, in which the coordination of the agents allows them to achieve something that they were not capable of individually. The opposite of this is concurrent interference, which models the case when multiple agents acting simultaneously decreases their capabilities, for example, two agents attempting, but failing, to simultaneously pass through a small doorway.

With sequential actions, it was assumed that, when one agent performs an action, the rest of the agents remain stationary, allowing the action to have its standard single-agent planning definition and effect even in a multiagent environment. With concurrent actions, it is assumed that all agents act simultaneously, meaning that a single agent may not know the outcome of its own action in advance. For example, if an agent attempts to lift a heavy object, then, with appropriately defined concurrent actions, this will only succeed if another agent chooses to help lift it at the same time. An action's complete effect is defined in terms of all the other actions to be performed simultaneously. This makes actions dependent on all other agents in the domain, and therefore makes planning in problems with concurrent actions much harder than planning in a sequential multiagent domain.

This in-built complexity leads to concurrent actions not being widely studied in multiagent planning, especially in multiagent planning that remains close to the classical planning paradigm. Most multiagent planning research is concerned with limiting the interaction problem as much as possible, while concurrent actions are inherently interactive and the size of the resulting joint action space can be prohibitively large. However, while they bring complexity, concurrent actions are required to describe certain interactions between agents.

This section introduces an efficient formalism for writing domains with concurrent actions that scales with the number of agents and size of the domain. It also introduces a series of multiagent planning benchmark domains that can be used to test an algorithms effectiveness over the different possible interaction types that can exist in multiagent planning problems. Finally, it discusses how to find plans in domains with concurrent actions.

6.2.1 Types of Multiagent Interaction in Planning Environments

The methods by which agents can interact in multiagent planning environments can belong to four different types. These are split into concurrent and non-concurrent interactions, along with the possibility that actions either model coordination or interference effects.

1. Non-concurrent Coordination: An agent may require another to provide a precondition required for a later action. For example, agent *a* needs to unlock a door before agent *b* can open it. In this case, performing an action allows another agent to perform an action later that they would otherwise not be able to. As a special case of non-concurrent coordination, an agent may (perhaps inadvertently) achieve another's goal.

2. Non-concurrent Interference: An agent may expect a precondition to be met at a specific point in its plan, but the precondition is removed by another agent, e.g. agent *a* may intend to pick up an object, but the object is moved by agent *b* before *a* can perform the pick-up action. As a special case of non-concurrent interference, an agent may (perhaps inadvertently) disrupt another's goal after it has been achieved (unless the problem exhibits goal-persistance).

3. Concurrent Coordination: Agents may need to coordinate simultaneous actions to achieve their intended effect. For example, when carrying a heavy object they need to move in the same direction. In this case, performing an action simultaneously allows the agent to do something that they could not do on their own.

4. Concurrent Interference: An agent's intended action may be adversely affected by another agent's simultaneous action. For instance if two agents simultaneously try to access a limited resource such as passing through a narrow doorway. In this case, an agent is prevented from achieving something that it could achieve on its own by a simultaneously attempted action from another agent.

The multiagent classical planning formalism of the previous chapters can only deal with interaction types 1 and 2. This chapter looks at the details of adding types 3 and 4 to the multiagent planning formalism. Naturally, this massively increases the complexity of the multiagent planning problem and, as such, it is not possible to solve such problems efficiently with current planning technology, except for the simplest problem instances.

6.2.2 Concurrent MMPTs

The main problem when creating a formalism for modelling concurrent actions, is how to deal with the size of the concurrent action space. Even with a very small number of agents and actions, the concurrent action space can be prohibitively large. This is because, with n agents, each with k possible actions, there are (ignoring preconditions) k^n possible concurrent actions that can be performed at any point during a plan.

One method in the literature for dealing with these concurrent actions is to define an admissibility function that returns whether of not a particular combination of individual actions is a valid concurrent action (see Section 2.5.4). This function defines $\Psi : A_1 \times \ldots \times A_n \rightarrow \{0,1\}$ from the Cartesian product of all agents' action sets to either 1, if the concurrent action is admissible, or 0, if it is not. While the domain of this function is the complete concurrent action set, this can potentially be mitigated by finding a compact definition based on its properties. Defining concurrency constraints (see Section 2.3.4), another approach in the literature, solves the problem of specifying the large domain of the admissibility function, by specifying them on the actions themselves.

Concurrency constraints are added to the PDDL definition of a problem and specify which other actions must, or must not, appear in the same concurrent action. For example, the preconditions of the `go_through_door(a, door1)` action would contain `(not (and (go_through_door(?agent, door)) (not (= (?agent, a)))))`, which says that it should not be the case that there is also an action of `go_through_door(?a, door)` where the variable `?agent` is instantiated to a different agent.

The drawback of this approach, is that it requires complete domain knowledge on behalf of the problem designer. Imagine that a large multiagent planning problem has been created and a new agent, with different capabilities to those already modelled, is to be added to the domain. The actions that this agent can perform need to be checked against every other action in the domain, even private actions of other agents. This means that the designer not only needs knowledge of, but also access to, every other action in the domain. By defining concurrency in terms of the objects of an MMPT, the method for describing concurrent actions constraints presented below allows for a concise formulation, local constraints and does not require knowledge of all other actions in the problem.

Definition A concurrent-MMPT, $\Pi = \langle V, \Phi, I, G, A, X, C \rangle$, is an MMPT with the additional element C, the set of concurrency constraints on objects in the system. Concur-

rency constraints are of the form $\langle o, c, l \rangle$ where o is an object in the PDDL problem definition and $c, l \in \mathbb{N} \cup \{\infty\}$.

The interpretation is that any time an object with a concurrency constraint appears in an action, then that action is bound by the concurrency constraint which specifies how many times that object must, or must not be used in a concurrent action in order for it to have the desired effect. For example, to model a doorway that can have at most one agent passing through at a time, the constraint $\langle \texttt{doorway}, 0, 1 \rangle$ is added to the PDDL definition, which specifies that if the doorway appears in an action, then it must not appear more than once. Similarly, a boat that requires two agents to operate it would have the concurrency constraint $\langle \texttt{boat}, 2, \infty \rangle$. If the boat had a maximum capacity of 10 agents then it would have the concurrency constraint $\langle \texttt{boat}, 2, 10 \rangle$.

This method for encoding concurrency constraints has the upside that it does not require full knowledge of the actions in the domain. Any agent, or new actions, that are to be added to the domain only need to be written to coincide with the intended interpretation of the objects in the domain, something that is required in any planning problem.

The downside of this approach is that it can require some slightly unintuitive use of objects when modelling a domain. Concurrency constraints on objects are actually related to the affordances of those objects, as opposed to the objects themselves. For example, when dealing with the concurrency constraint $\langle \texttt{doorway}, 0, 1 \rangle$, it is the affordance of the doorway to allow agents to pass through it that has the concurrency constraint, whereas the affordance of the doorway to allow agents to perceive its state (for example) may not be bound by any concurrency constraints. When objects have multiple affordances it is expected that an object exists in the PDDL definition of the problem for each possible affordance. In the doorway case this would mean having an object, with associated concurrency constraint $\langle \texttt{doorway-pass-through}, 0, 1 \rangle$ as well as a separate `doorway-perceived` object which does not have a concurrency constraint.

Returning to the formalism, the state transition function is now defined in terms of *concurrent actions*. A concurrent action $\bar{a} = \langle a_1, \ldots, a_n \rangle$ is an ordered list of actions, one for each agent ϕ_i. The notation $a_1 \in \bar{a}$ is used to show that a_1 is an action that is a component of $\bar{a}$. Let $param(a) = pre(a) \cup eff(a)$ and $obj(a) = \{o \in O \mid o \text{ appears in } param(a)\}$. Then, define $pre(\bar{a}) = pre(a_1) \cup \ldots \cup pre(a_n)$, and similarly $obj(\bar{a}) = obj(a_1) \cup \ldots \cup obj(a_n)$.

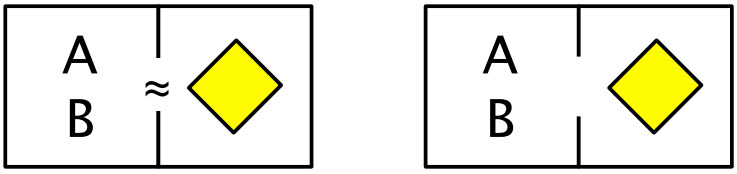

Figure 6.1: Boats (left) and Doors (right). Agent locations are shown using capital letters, target locations using shaded diamonds. A boat connection is represented by ≈ while a door connection by a narrow gap between grid spaces.

A concurrent action needs to satisfy:

$$\forall(o,c,l) \in C : o \in obj(\bar{a}) \implies c \leq |\{a \in \bar{a} | c \text{ appears in } param(a)\}| \leq l.$$

In other words, a concurrent object (o,c,l) *must* be used by at least c agents, if it is used at all, and can only be used by (up to) l agents in a single concurrent action. For any object o that violates the above constraint, any actions that contain that object are replaced with a `noop` action that has no preconditions and no effects.

A further constraint, that:

$$eff^{+}(\bar{a}) \cap eff^{-}(\bar{a}) = \emptyset$$

is added so that two actions cannot simultaneously attempt to add and delete the same fluent. Such actions are also replaced with `noop` actions when it comes to applying the action. This approach coincides with the interpretation of objects with concurrency constraints representing the affordances of objects, which, in this case, are being accessed in a contradictory way. An application of $\bar{a}$ in state S will result in a new state $S \backslash eff^{-}(\bar{a}) \cup eff^{+}(\bar{a})$ after changing any individual actions to `noop` that violate the concurrency constraints.

6.2.3 Multiagent Planning Domain Set

It was argued in Section 2.2, that the multiagent planning community is in need of a benchmark testing problem set, and a clear formulation of the multiagent planning problem, in order to help focus the research area. Now that a formalism for concurrent actions has been introduced, it is possible to describe a set of simple multiagent planning problems that can be used to test a multiagent algorithms performance over each of the four different possible interaction types. These are designed to be modular (they can be

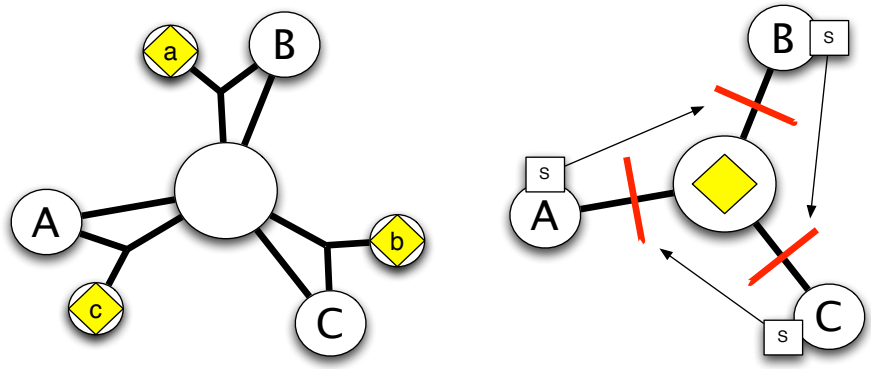

Figure 6.2: Bridges (left) and Switches (right). Agent locations are shown using capital letters, target locations using shaded diamonds (labelled with agent names, if not common to all agents).

combined to test multiple interaction types at once) and easily adapted to increase the number of agents and size and complexity of the problem.

- **Boats (concurrent coordination):** In the Boats domain, moving between two locations requires multiple agents travelling at the same time (an agent cannot operate a boat on its own). Therefore, $(\mathtt{b}, k, \infty) \in C$ for all boats $\mathtt{b}$. The basic problem, shown in Figure 6.2, contains two locations connected by a boat and $k = 2$. Each agent has to travel from the starting location to the goal location. This problem can be easily extended by varying the number of agents, the value of k or the size and complexity of the domain.

- **Doors (concurrent interference):** This domain models the simple coordination problem of two agents trying to fit through the same narrow doorway. Agents can not go through the doorway simultaneously. This is modelled by including $(\mathtt{d}, 0, k) \in C$ for all doors $\mathtt{d}$. The structure of the basic problem, shown in Figure 6.2 is the same as for the Boats domain and $k = 1$. Again, this can be easily modified to increase the number of agents, the value of k and the size and complexity of the domain.

- **Bridges (non-concurrent interference):** In this domain, the connections between locations are badly constructed bridges that fall down after a single crossing. As only non-concurrent interference is to be tested, agents may simultaneously cross the same bridge if this is in a simultaneous actions setting. Figure 6.1 shows a basic problem instance. Each agent has to cross to the central island and then

out to its goal. There are two possible bridges that can be used to cross to the central island, one of them has no effect on the other agents while the other stops another agent from reaching its goal. It is easy to add further agents to the domain by increasing the number of orbiting islands in the diagram, and this, unlike with previous concurrent action approaches, only requires a linear increase in the size of the domain definition.

- **Switches (non-concurrent coordination):** This is the counterpart to the bridge domain; pathways are created instead of destroyed. Along with a `move` action, this domain also requires a `push-switch` action which opens up a passage for another agent. The basic problem, shown in Figure 6.1, has the same cyclic structure as the Bridge domain and can be extended in a similar manner.

PDDL definitions of the move action in these domains are shown in Figure 6.3. It can be seen that the definitions and concurrency constraints required are relatively simple. If another agent needs to be added with its own move action, perhaps because it has extra conditions on moving than the other agents, then this can be easily done as long as the action references the object that represents the doors affordance of being passed through.

Note that the full domain definitions will include 'noop' actions for each agent along with `reported` actions with the goal specifications being that the respective agents have reported at their goal locations. Adding the 'noop' action ensures that agents can always coordinate concurrent actions if they need to, and adding the `reported` action ensures that each domain has the 'goal persistence' property that is introduced in the next section.

As the domains are designed to be modular, it is possible to combine them, which forms the Maze domain, of which a possible problem instance is shown in Figure 6.4. This domain contains all the possible interaction types; but, of course, it is possible to use any combination of them.

6.2.4 Solving Problems with Concurrent Action Constraints

Solving problems with concurrent actions would ideally be performed by distributed agents acting individually. This would utilise the locality provided by the definition of concurrent actions and also incorporate the multiagent nature of this extension. However, such methods are beyond the classical planning style centralised approach

Boats

```
(:action move-boat
:parameters (?a - agent ?b - row_boat ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?b ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x)))
)
```

Concurrency constraint: $\langle$row_boat_object$,2,\infty\rangle$

Doors

```
(:action move-door
:parameters (?a - agent ?d - pass_door ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?d ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x)))
)
```

Concurrency constraint: $\langle$pass_door_object$,0,1\rangle$

Bridges

```
(:action move-bridge
:parameters (?a - agent ?b - cross_bridge ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?b ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x)) (not (connected ?b ?x
?y)))
)
```

Switches

```
(:action press-switch
:parameters (?a - agent ?s - switch ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (at ?s ?x) (switch ?s ?x ?y))
:effect (connected ?x ?y)
)
(:action move
:parameters (?a - agent ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x)))
)
```

Figure 6.3: Action and Concurrency Constraint Definitions for the Multiagent Planning Domains.

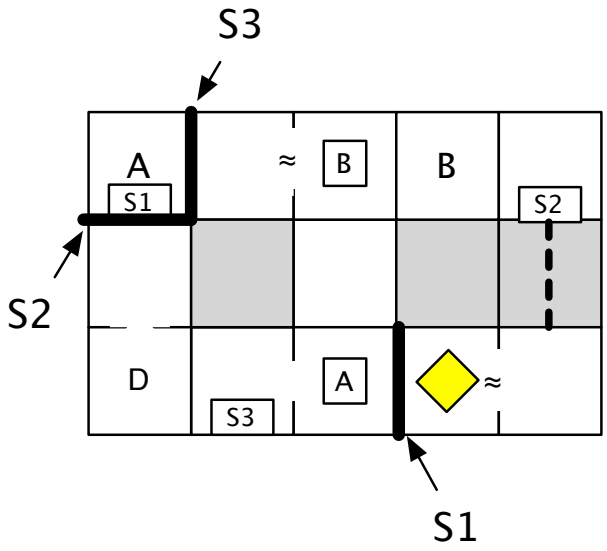

Figure 6.4: An example problem for the maze domain. This contains each of the possible agent interactions found in the Boats, Doors, Bridges and Switches domains. The thick black lines represent connections that are originally locked until their appropriate switches have been pushed.

taken in this thesis and, indeed, beyond the current level of multiagent planning research for any significantly sized problems. Instead, this section shows how solutions could be found with a centralised method which could potentially be used as a comparison for any distributed multiagent approaches that appear as the multiagent planning research area matures.

The solution method translates the problem into a simple-time representation with equality and presupposes that each operator definition contains a reference to an agent variable. With the operator's ground instances belonging to the agent its agent variable becomes ground to. Simple-time planning problems represent durative actions (actions with durations that are used by temporal planning approaches) by turning them into separate start and end actions.

As a first step, all concurrent action constraints are removed from the object file and all action's preconditions are appended with a new parameter `(new_action_available)` which is also added to the initial state. Then, each operator definition that contains a concurrent action constraint is split up into a begin action, and a number of join actions and end actions depending on the constraint definition.

Imagine that the action to be translated is the move-door action from the doors

domain, except, to make the example more illustrative, the concurrency constraint is replaced with $\langle$`pass_door_object`$,2,3\rangle$ which says that the door must be used by at least two, and at most three, agents. For reference, the original actions is:

```
(:action move-door
:parameters (?a - agent ?d - pass_door ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?d ?x ?y))
:effect (and (at ?a ?y) (not (at ?a ?x)))
)
```

First, the begin action is created with the same parameters:

```
(:action begin-move-door
:parameters (?a - agent ?d - pass_door ?x - loc ?y - loc)
```

Then, the preconditions are added, and they have already been appended with `(new_action_available)` that is used to stop any other actions from being started until this one is completed.

```
:precondition (and (at ?a ?x)
      (connected ?d ?x ?y) (new_action_available))
```

After this, the effects are added, except that they no longer include the normal action's effect but instead a proposition designed to block all other actions in the domain. They are also appended with a new parameter that is the action's name appended with `-ing` along with the agent object that represents that the agent is currently performing the related action.

```
:effect (and (at ?a ?y) (not (at ?a ?x))
    (not (new_action_available)) (move-door-ing(?a)))
)
```

At this point a number of join actions are created equal to the $l-1$ from the concurrent action constraint; which in this case is $3-1=2$. The join-actions preconditions are the same as the original action, except that a number of agents must be action`-ing` equal to how many join actions have been created so far. Extra parameters are added for each agent that is action`-ing`. Furthermore, is is important that the agents in the action are distinct so a `(not (= ?a ?b))` precondition has to be added between every possible agent pair.

```
(:action join-move-door
:parameters (?a - agent ?b - agent ?d - pass_door ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?d ?x ?y)
    (move-door-ing ?b) (not (= ?a ?b)))
```

The effects of this new action will add another action-ing proposition and again omit their original contents.

```
:effect (move-door-ing ?a)
)
```

The second join action will be as follows:

```
(:action join-move-door
:parameters (?a - agent ?b - agent ?c - agent
                          ?d - pass_door ?x - loc ?y - loc)
:precondition (and (at ?a ?x) (connected ?d ?x ?y)
    (move-door-ing ?b) (move-door-ing ?c)
     (not (= ?a ?b)) (not (= ?a ?c)) (not (= ?b ?c)))
:effect (move-door-ing ?a)
)
```

Note that the number of `(= ?agent1 ?agent2)` constraints in each action is $\frac{n(n-1)}{2}$ where n is the number of agents in that agents parameters so this method is not feasible for problems with large numbers of agents.

To finish the action, a number of end actions are added, equal to $l - c$, one for each possible number of agents that can be involved in the action when it is ended. For example, if there are just two agents using the joint action, then this would be:

```
(:action move-door
:parameters (?a - agent ?b - agent ?d - pass_door ?x - loc ?y - loc)
```

As a precondition, all agents involved are required to already be performing the action.

```
:precondition (and (move-door-ing ?a) (move-door-ing ?b))
```

The effects of the action contain the normal action effects for each agent that is to perform the action as well as the proposition that allows all other actions to be used again.

```
:effect (and  (at ?a ?y) (not (at ?a ?x))
                    (at ?b ?y) (not (at ?b ?x))
            (new_action_available))
)
```

After converting all the actions in the domain like this it is possible to solve the problem using a standard planner. The output can then be converted into a concurrent action plan by converting sequences of begin, joint and end-actions into concurrent actions with the appropriate agents. This would include 'noop' actions for each agent not involved in a particular action. The solution quality can then be improved by simply combining two actions a_1 and a_2 that are next to each other in the plan if they contain disjoint sets of agents and the action's effects do not interact. Due to the complexity of concurrent actions, this translation process creates a problem file that is exponential in the number of agents in the problem and, it will not be until further advances have been made in multiagent planning, that a practical planning method can be applied to these kind of domains.

6.3 Summary

This chapter considered the game theoretic issues that arise from agents having individual goals and preferences over the plans that achieve them. It was argued that because the ability to look ahead is so fundamental to planning, solution concepts in strategic multiagent planning problems should take into account farsightedness. To this end, a new solution concept was introduced and a class of problems defined for which it is possible to find stable solutions using existing planning technology.

This chapter discussed the extension of multiagent classical planning to include concurrent action constraints. These allow for concurrent actions to be defined such as the joint lifting of a heavy object or the inability of multiple agents to simultaneously pass through a confined space. A method was introduced for defining multiagent planning problems in PDDL that is efficient in the size of the problem definition in relation to the number of agents. This involved associated concurrency constraints with the affordances of objects in the domain.

Considering concurrent actions allowed for the four possible methods of agent interaction to be clearly defined. A planning domain was then introduced for each of the possible interactions. These domains have the property that they are easily combinable

so that algorithms can be tested against specific combinations of interaction types. The domains can also be easily modified to increase their size, or the number of agents involved.

Chapter 7

Conclusion

This thesis has covered multiple topics related to multiagent classical planning. It introduced an automated decomposition algorithm along with a heuristic multiagent planning algorithm. It also discussed two extensions of multiagent classical planning: concurrent actions and strategic considerations. The evaluation section showed that the heuristic planning algorithm, combined with the decomposition algorithm, effectively finds and exploits inherent multiagent structure and improves on state-of-the-art methods in the field.

7.1 Future Work

There is an almost endless amount of future work that can be done in the area of multiagent planning. As the problem area becomes more defined and there is more work to build on, it will be possible to move further from the classical planning assumptions. Hopefully, the preliminary work presented in the extensions chapter will be able to form the basis for future multiagent planning systems and approaches.

7.1.1 Decomposition Algorithm

The effectiveness of the decomposition algorithm can be utilised in multiple ways. It can be used to check for underlying multiagent structure in planning problems. This can lead the way for new domains to be defined for use in the wider planning community. It can also be used to group domains by their inherent structures and to study how this structure is related to their complexity under different planning approaches. The decomposition algorithm provides an easy method for writing multiagent planning problems that can

be used by multiagent planning approaches. Domains can simply be written in the well-known and well-understood single-agent PDDL and then converted to MMPTs with the decomposition algorithm. This allows for easier access to multiagent planning problems for the multiagent planning community. So, one direction for future work is to use the decomposition algorithm as a stand-alone process for creating multiagent planning problems from standard PDDL-input.

While the decomposition returned verifiably multiagent decompositions, the exact nature of these, and the detailed limitations and uses of the algorithm remain for future work. It may be possible to create functionally identical problem definitions for multiagent domains that 'trick' the algorithm into not returning the expected decomposition. While this is not necessarily a drawback of the algorithm, as the domains that formed the basis of the evaluation can be assumed to be representative of how planning domains are written, this could be an interested area for future research in order to develop a deeper understanding of the techniques. This could potentially lead to improving the algorithm so that it can find different types of multiagent decomposition, or even so that it can find other useful structures that are not necessarily multiagent in nature.

It was shown how the causal graphs used by the decomposition algorithm need to be modified slightly by those used by LAMA. Whenever an action created two-way edges (both (v, v') and (v', v)) then these needed to be removed from the graph in order for the algorithm to have the intended properties. The results of applying this new type of causal graph to existing planning approaches that utilise causal graphs could prove to be interesting. It may be that this alteration is also useful in a wider context.

7.1.2 ADP

The obvious direction to extend the ADP algorithm is to try to improve its performance across multiagent planning problems. The version presented in this thesis chose agent subgoals by picking the agent with the most achievable goals in its single-agent subproblem and searching for them all at once. This is an intuitive approach and much more effective than the method at the other end of the spectrum that picks only a single subgoal at a time because it combines search for achievable goals together therefore saving time over searching for them all individually. However, there is a largely unexplored area in between these two approaches where more sophisticated methods can be used for choosing the goal decomposition. Perhaps the nature of the goals themselves, or their relations in the causal graphs could be analysed to group them together in an intelligent

manner. This could then leverage the advantages of both ends of the spectrum.

A further area of extension for ADP is to focus on more efficient backtracking. Sometimes, the algorithm will reach a global dead-end, but this will not be noticed until the next goal decomposition step is performed. In complex problems with lots of dead ends, such as Floortile, this leads to ADP having less of an improvement over single-agent approaches. Adding in naive methods for checking for dead ends more often just leads to slowing down the heuristic calculation time too much. More sophisticated methods could be implied, including analysis of which influencing actions may cause problems. Whenever an influencing action is added to the search tree, it may be possible to find a way to check for global dead-ends by comparing the action to the relaxed planning graphs generated in the most recent goal decomposition stage.

There is also the possibility that ADP can be integrated with other heuristic search methods. Currently, ADP uses relaxed planning graphs and FF-heuristic calculations, but does not exploit any other planning methods. It is an open question how other planning techniques can be combined with the ideas used in ADP, but, given the improvement of ADP over FF, it is natural to conjecture that similar improvements could be made by applying the techniques of ADP to other planning methods.

7.1.3 Moving Away From Multiagent Classical Planning

Moving away from multiagent classical planning leaves a much broader research area, but one that is much more complex and for which it is harder to provide definitive results. It is this authors opinion that the future breakthroughs in both strategic multiagent planning and planning concurrent actions will come from domain specific work based on real-world problems. This work would follow a very different research agenda to the domain independent planning assumed throughout this thesis.

This thesis argued that there is a dichotomy between game theoretic approaches and planning. However, there are many individual applications that could be modelled as planning problems, yet involve self-interested agents. For these domains, the assumptions that can be made about the agents will come from the problem being modelled. For specific instances, it should then be possible to utilise work from game theory that corresponds with the assumptions. This will allow for progress to be made in strategic multiagent planning, albeit, for domain specific work. It is likely that a lot of work of this type will be required before more domain independent work on strategic planning will be fruitful.

The most promising direction for multiagent planning, that will allow for strategic planning and planning with concurrent actions to be studied in a domain independent manner, is distributed multiagent planning. The ADP algorithm is implemented as a centralised planning algorithm, but there are large parts of the process that can be distributed due to the fact that most of the planning process is performed by a single agent at a time. Not only could distributing the algorithm lead to faster planning times, but it could allow for easier research into distributed planning with issues such as agent privacy and communication being explored. The strategic and concurrent action extensions fit naturally in a distributed planning setting, and once work on this area becomes more prevalent, these areas could start to see rapid progress. The way that concurrent action constraints are presented in this thesis allows for the local problem of the agents to be solvable without requiring knowledge of all the other agents in the domain and could be applicable in this area.

7.2 Contributions Revisited

This section reviews the contributions of the thesis now that the algorithms and results have been presented.

- **Formalisation of MMPTs:** MMPTs were introduced as an extension of MPT's to multiagent planning. Agents were defined as collections of variables that represented their internal states and this interpretation of agents was later shown to be effective at picking out what it means to be an agent across many planning domains. This represented an important shift in focus from agents as interacting sets of actions to agents as entities that interact through an environment.

- **Agent Decomposition Algorithm:** An algorithm was introduced that can find agent decompositions (valid MMPTs) given single-agent classical planning problems as input. This algorithm was shown to find the expected decompositions and even return more detailed decompositions on all domains tested. This confirmed that the approach of defining agents based on their internal states presented with the formalisation of MMPTs was justified.

- **ADP, an Agent Decomposition Planner** ADP, a heuristic multiagent planning algorithm, was shown, when combined with the decomposition algorithm, to greatly improve on state-of-the-art planners on multiagent planning domains.

Unlike most previous multiagent planning approaches that focus on loosely coupled domains, ADP showed improved performance even on domains that require complex interaction between the agents.

- **Extensions** A new solution concept for coalition planning games was introduced that was shown to have more intuitive results than the solution concept taken from the literature. This was because the new solution concept took into account farsightedness, which models the ability to look ahead which is, of course, fundamental in planning. A subclass of strategic planning problems, safe-domains, was defined that can be solved using modifications of existing planning heuristics. While restrictive, this subclass is broad enough to include the parcel delivery domain introduced in Section 1.2.2. A method for defining concurrent action constraints was also presented, that allows for problem domains to be specified without knowledge of the actions belonging to other agents. This method was later displayed in action in the canonical multiagent planning domains where it could be seen that it allowed for efficient problem descriptions. Finally, a set of planning domains was introduced that covers the four different ways in which agents can interact.

7.3 Summary

The main contributions of this thesis were an automated decomposition algorithm and a heuristic multiagent planning algorithm that combined can solve any classical planning problem with an inherent multiagent structure. The decomposition algorithm was shown to find multiagent decompositions with a strong correlation to the multiagent structure expected to be found in classical planning problems. In fact, the decomposition algorithm managed to find more detailed decompositions than expected by linking agents with parts of the domain that only they have control over.

The planning algorithm solved multiagent domains more efficiently than existing state-of-the-art planners. The improvement in planning times was not constrained to domains with little coordination but the algorithm was also very effective in domains that require high levels of coordination. This algorithm reduced the complexity of the multiagent decompositions by solving the coordination problem in a relaxed version of the search space. Effective methods were introduced for combining agent's heuristic values for their individual subproblems so that the full problem space is still covered.

Further discussion presented a solution concept that has more intuitive consequences on that in the literature and a subclass of domains for which solutions can be found using existing planners. It was shown how to define concurrent actions based on object's affordances. This allowed for efficient writing of multiagent domains, even with large numbers of agents. Finally, a set of multiagent planning domains was introduced that can potentially be used to test the effectiveness of future multiagent planning approaches.

Appendix A

Further ADP Results

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0	0.04	0.18	38	**22**	25	**265**	3140	4630
0	0.04	0.14	33	**16**	18	**572**	2850	3630
0.01	0.09	0.4	29	**23**	29	**1070**	3560	6890
0.39	**0.24**	1.17	60	**50**	**50**	10600	**7710**	19200
0.01	0.15	0.02	**41**	65	57	**97**	2820	266
2.85	43.8	6.3	158	185	**138**	**34900**	346000	35200
2.75	1.91	**0.44**	**109**	171	112	23400	11100	**1720**
3.95	6.62	9.32	134	**123**	183	**25800**	30000	30200
9.22	**7.89**	9.15	**179**	376	284	60700	29500	**25700**
8.18	8.26	**3.71**	**157**	160	175	32600	20700	**5530**
27.4	68.9	30.5	**867**	1150	1030	189000	451000	**125000**

Table A.1: Planning times, cost and the number of states evaluated for problems from the Driverlog domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.08	2.31	16.9	101	67	**65**	**2110**	202000	520000
0.03	1.68	22	84	**64**	**64**	**861**	155000	646000
3.06	21.5	29.1	187	93	**71**	**60100**	1370000	727000
0.44	31.6	229	152	**111**	117	**9660**	2180000	6090000
32.2	–	–	**211**	–	–	**383000**	–	–
3.61	57	297	524	335	**317**	**72700**	3900000	7980000

Table A.2: Planning times, cost and the number of states evaluated for problems from the Floortile domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0	0.06	0.12	**23**	106	109	**24**	529	824
0	0.18	0.3	**28**	136	124	**29**	853	1050
0.01	0.34	0.51	**23**	119	111	**24**	1230	1380
0.01	0.27	0.27	**22**	110	108	**23**	718	660
0	0.53	1.04	**31**	161	155	**32**	1670	2320
0.01	0.88	1.56	**36**	220	196	**37**	2440	3130
0.01	1.24	1.3	**36**	184	180	**37**	1320	1700
0.01	0.4	0.69	**27**	134	137	**28**	609	849
0.01	1.42	2.4	**35**	179	191	**36**	1700	2560
0.01	0.84	2.01	**35**	172	174	**36**	1180	2140
0.02	2.12	4.27	**43**	221	215	**44**	1900	2910
0.02	3.15	7.66	**44**	212	198	**45**	1330	2570
0.03	2.86	6.6	**48**	226	224	**49**	1650	2680
0.03	3.77	9.2	**50**	254	243	**51**	1800	2870
0.04	8.23	30	**61**	299	275	**62**	3480	7200
0.06	10.5	27.8	**58**	294	272	**59**	4120	5350
0.27	36	94.2	**518**	2620	2520	**531**	23500	36000

Table A.3: Planning times, cost and the number of states evaluated for problems from the Tpp domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.11	6.34	1.42	1540	2390	**1270**	**381**	4940	706
0.11	9.68	1.82	1570	2980	**1140**	**323**	5560	695
0.13	–	8.04	3750	–	**3640**	**309**	–	1900
0.05	–	1.83	**3170**	–	4750	**160**	–	1570
0.09	8.86	4.89	**4030**	5540	5510	**333**	6950	3250
0.14	–	11.9	5350	–	**4650**	**483**	–	4390
0.17	–	48.1	**5280**	–	7480	**461**	–	15200
0.22	7.92	8.34	2000	2790	**1740**	**590**	3410	2540
0.33	–	81.9	**4310**	–	7140	**795**	–	19000
0.31	–	29.6	**5670**	–	6440	**807**	–	5910
0.2	30.5	4.78	2380	2750	**1610**	**451**	9960	1080
0.22	36.9	19.9	2560	3690	**2410**	**553**	11200	4940
0.28	–	19.4	3500	–	**2510**	**652**	–	3930
2.02	–	–	**4620**	–	–	**2300**	–	–
0.48	–	197	**4220**	–	9130	**527**	–	18300
1.02	–	–	**5240**	–	–	**1060**	–	–
1.41	–	–	**6630**	–	–	**1110**	–	–
0.78	–	–	**5120**	–	–	**639**	–	–
1.73	–	–	**6960**	–	–	**1240**	–	–
1.13	–	–	**5500**	–	–	**770**	–	–
0.95	100	41.1	14100	20100	**13700**	**2630**	42000	13200

Table A.4: Planning times, cost and the number of states evaluated for problems from the Transport domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.02	0.03	0.11	**40**	**40**	44	**71**	207	267
0.03	0.07	0.15	**39**	51	53	**80**	222	376
0.06	0.1	0.26	**52**	**52**	56	**115**	235	457
0.06	0.17	0.56	**71**	**71**	75	**219**	522	766
0.09	0.27	0.47	**64**	80	78	**262**	754	490
0.11	1.73	1.26	91	**83**	90	**237**	3930	1150
0.13	0.54	2.18	**91**	95	94	**289**	990	1380
0.5	2.91	4.99	**448**	472	490	**1270**	6860	4880

Table A.5: Planning times, cost and the number of states evaluated for problems from the Zenotravel domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.02	0.36	0.25	389	471	**373**	**130**	2150	1040
0.44	–	41.3	**636**	–	785	**5900**	–	58300
0.05	–	0.36	700	–	**442**	**330**	–	857
0.07	–	37.3	**800**	–	1030	**454**	–	36300
0.09	–	146	**1090**	–	1210	**453**	–	120000
0.15	–	–	**1080**	–	–	**919**	–	–
0.15	–	–	**1460**	–	–	**498**	–	–
0.24	–	–	**1330**	–	–	**888**	–	–
0.17	–	–	**1320**	–	–	**538**	–	–
0.2	–	–	**1520**	–	–	**602**	–	–
0.29	–	–	**1450**	–	–	**831**	–	–
0.23	–	–	**1650**	–	–	**735**	–	–
0.32	–	–	**1910**	–	–	**917**	–	–
4.58	–	–	**2360**	–	–	**17400**	–	–
0.66	–	–	**3000**	–	–	**1700**	–	–
0.61	–	–	**2000**	–	–	**1290**	–	–
0.02	0.36	0.25	389	471	**373**	**130**	2150	1040

Table A.6: Planning times, cost and the number of states evaluated for problems from the Elevators domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.01	0.27	2.22	**21**	23	22	**27**	163	1360
0	0.35	0.37	**12**	**12**	**12**	**13**	60	90
0	0.34	0.32	**14**	17	17	**19**	59	91
0.01	0.32	0.17	14	**11**	**11**	**16**	24	31
0.01	0.16	0.06	9	**8**	**8**	**11**	18	22
0.01	0.3	0.68	22	**21**	24	**23**	136	334
0.01	0.14	0.04	11	**8**	**8**	14	**13**	18
0.01	0.39	0.7	**8**	13	15	**10**	80	165
0.01	0.26	0.24	**8**	16	11	**11**	169	177
0	0.14	0.13	**11**	13	13	**13**	73	103
0.07	1.84	4.11	**93**	100	99	**112**	603	2110

Table A.7: Planning times, cost and the number of states evaluated for problems from the Mystery domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.01	201	**0.01**	**56**	60	**56**	**153**	1.07E7	175
1.01	**0.01**	0.02	93	69	**66**	12000	**209**	234
45.2	**0.02**	0.07	106	104	**102**	336000	**373**	580
0.37	**0.03**	0.09	129	**106**	112	608	**476**	776
0.17	**0.07**	0.19	148	**133**	149	**598**	669	1140
3.12	**0.07**	0.21	174	**148**	151	20900	**692**	1260
1.06	**0.11**	0.39	193	175	**172**	2720	**1470**	2720
0.59	**0.15**	0.39	**163**	171	177	**831**	1470	2120
0.72	**0.13**	0.49	201	**195**	**195**	2700	**1680**	3010
2.04	**0.23**	0.53	210	205	**188**	4410	**2500**	2970
6.96	**0.08**	0.19	168	**156**	157	35900	**904**	1090
0.29	**0.12**	0.47	200	**185**	193	1360	**1210**	2570
1.05	**0.19**	0.37	235	226	**220**	3350	**1830**	1870
1.63	**0.15**	0.51	229	207	**202**	2450	**1090**	2060
0.51	**0.21**	0.56	240	**216**	**216**	**931**	1620	2370
2.77	**0.3**	1.08	252	249	**247**	6340	**2660**	4010
0.59	**0.17**	0.57	244	221	**215**	2530	**1300**	2260
2.79	**0.4**	1.34	282	274	**270**	7150	**3800**	6580
40.8	**0.6**	1.98	315	289	**284**	46100	**5190**	8220
112	204	**9.46**	3640	3390	**3370**	487000	1.08E7	**46000**

Table A.8: Planning times, cost and the number of states evaluated for problems from the Pathways domain

Search Time (s)			Plan Cost			States Evaluated		
ADP	FF	LAMA	ADP	FF	LAMA	ADP	FF	LAMA
0.21	0.09	**0.01**	23	**22**	**22**	5020	2980	**297**
0.32	0.12	**0.02**	51	**23**	**23**	8520	2870	**270**
–	0.16	**0.04**	–	**20**	**20**	–	6980	**863**
0.54	**0.16**	0.6	91	**28**	30	13600	**4470**	8970
0.35	**0.15**	0.56	45	**22**	24	7480	**3010**	6490
2.88	**0.3**	1.14	114	**24**	71	36000	**3730**	8390
10.8	**0.7**	1.89	118	**33**	45	114000	**6430**	10800
–	36.8	**3.21**	–	**43**	79	–	243000	**13600**
117	–	–	**247**	–	–	**746000**	–	–
–	**50.9**	209	–	**63**	85	–	**111000**	335000
15.1	**1.52**	4.22	442	**152**	215	185000	**23500**	35200

Table A.9: Planning times, cost and the number of states evaluated for problems from the Storage domain

Bibliography

Bäckström, C. and Nebel, B. (1995). Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655.

Bonet, B. and Geffner, H. (1999). Planning as heuristis search: New results. In Biundo, S. and Fox, M., editors, *Proc. 5th European Conf. on Planning*, pages 359–371, Durham, UK. Springer: Lecture Notes on Computer Science.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33.

Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. *Theoretical Aspects of Rationality and Knowledge*, 6:195–210.

Boutilier, C. and Brafman, R. (2001). Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, 14:105–136.

Bowling, M., Jensen, R., and Veloso, M. (2002). A formalization of equilibria for multiagent planning. *Association for the Advancement of Artificial Intelligence: Workshop on Planning with and for Multiagent Systems*.

Brafman, R. and Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. *International Conference on Automated Planning and Scheduling*, 18:28–35.

Brafman, R., Domshlak, C., Engel, Y., and Tennenholtz, M. (2009). Planning games. *International Joint Conference on Artificial Intelligence*, 21:73–78.

Brenner, M. (2003). A multiagent planning language. *International Conference on Automated Planning and Scheduling: Workshop on PDDL*.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204.

Crosby, M. and Rovatsos, M. (2011). Heuristic multiagent planning with self-interested agents. *Autonomous Agents and Multi-Agent Systems*, 22:1213–1214.

Crosby, M., Rovatsos, M., and Petrick, R. (2013). Automated agent decomposition for classical planning. *International Conference on Automated Planning and Scheduling*, 23:46–54.

de Weerdt, M. M. and Clement, B. J. (2009). Introduction to planning in multiagent systems. *Multiagent and Grid Systems An International Journal*, 5(4):345–355.

de Weerdt, M. M., Mors, A. T., and Witteveen, C. (2005). Multi-agent planning: An introduction to planning and coordination. Technical report, European Agent Summer School.

Durfee, E. and Lesser, V. R. (1991). Partial global planning: A coordination framework for distributed hypothesis formation. *Institute of Electrical and Electronics Engineers: Transactions on Systems, Man, and Cybernetics*, 21:1167–1183.

Edelkamp, S. and Helmert, M. (1999). Exhibiting knowledge in planning problems to minimize state encoding length. *European Conference on Planning*, 5:135–147.

Ephrati, E. and Rosenschein, J. (1993a). Multi-agent planning as the process of merging distributed sub-plans. *International Workshop on Distributed Artificial Intelligence*, 12:115–129.

Ephrati, E. and Rosenschein, J. S. (1993b). Multi-agent planning as a dynamic search for social consensus. *International Joint Conference on Artificial Intelligence*, 13:423–431.

Ephrati, E. and Rosenschein, J. S. (1994). Divide and conquer in multi-agent planning. *Association for the Advancement of Artificial Intelligence*, 12:375–380.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Georgeff, M. (1983). Communication and interaction in multi-agent planning. *Association for the Advancement of Artificial Intelligence*, 3:125–129.

Gibbons, R. (1992). *A primer in game theory*. Harvester Wheatsheaf.

Grosz, B. and Kraus, S. (1996). Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357.

Haslum, P. and Geffner, H. (2000). Admissible heuristics for optimal planning. *Association for the Advancement of Artificial Intelligence*, 17:140–149.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Hoffmann, J. and Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302.

International Planning Competition (2008). http://ipc.informatik.uni-freiburg.de/. Web Site.

Jensen, R. M., Veloso, M. M., and Bowling, M. H. (2001). OBDD-based optimistic and strong cyclic adversarial planning. *European Conference on Planning*, 6:265–276.

Jonsson, A. and Rovatsos, M. (2011). Scaling up multiagent planning: A best-response approach. *International Conference on Automated Planning and Scheduling*, 21:114–121.

Jonsson, P. and Bäckström, C. (1995). Incremental planning. *European Workshop on Planning*, 3:79–90.

Jonsson, P. and Bäckström, C. (1998). State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1–2):125–176.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302.

Kovacs, D. L. (2012). A multi-agent extension of PDDL3.1. *International Conference on Automated Planning and Scheduling: Workshop on the International Planning Competition*, 22-3:19–27.

Lansky, A. (1991). Localized search for multiagent planning. *International Joint Conference on Artificial intelligence*, 12:252–258.

Larbi, R. B., Konieczny, S., and Marquis, P. (2007). Extending classical planning to the multi-agent case: A game-theoretic approach. *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, 9:731–742.

McDermott, D. (2000). The 1998 ai planning systems competition. *Artifical Intelligence Magazine*, 21(2):35–55.

Morgenstern, O. and Von Neumann, J. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Moses, Y. and Tennenholtz, M. (1995). Multi-entity models. *Machine Intelligence*, 14:918–923.

Nash, J. F. (1951). Non-cooperative games. *Annals of Mathematics*, 54(2):286–295.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco.

Newell, A. and Simon, H. (1963). GPS: a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*, pages 279–293. McGraw-Hill.

Nissim, R., Apsel, U., and Brafman, R. I. (2012). Tunneling and decomposition-based state reduction for optimal planning. *European Conference on Artificial Intelligence*, 20:624–629.

Nissim, R., Brafman, R. I., and Domshlak, C. (2010). A general, fully distributed multi-agent planning algorithm. *International Conference on Autonomous Agents and Multiagent Systems*, 8:1323–1330.

Osborne, M. J. and Rubinstein, A. (1994). *A Course in Game Theory*. The MIT Press.

Parsons, S. and Wooldridge, M. (2002). Game theory and decision theory in multi-agent systems. *International Conference on Autonomous Agents and Multi-Agent Systems*, 5(3):243–254.

Ray, D. (2007). *A Game-Theoretic Perspective on Coalition Formation*. Lipsey Lectures Series. OUP Oxford.

Rosenschein, J. S. (1986). *Rational Interaction: Cooperation Among Intelligent Agents*. PhD thesis, Stanford University.

Rosenschein, J. S. and Genesereth, M. R. (1985). Deals among rational agents. *International Joint Conference on Artificial intelligence*, 9:91–99.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Sacerdoti, E. D. (1973). Planning in a hierarchy of abstraction spaces. *International Joint Conference on Artificial Intelligence*, 3:412–422.

Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Weiss, G., editor (1999). *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA.

Weld, D. S. (1999). Recent advances in ai planning. *Artifical Intelligence Magazine*, 20:93–123.

Williams, B. C. and Nayak, P. P. (1997). A reactive planner for a model-based executive. *International Joint Conference on Artificial Intelligence*, 15(2):1178–1185.

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.