

A Comparative Study of Approaches to Multi Agent Planning

Dave Gurnell

A thesis submitted
to the University of Birmingham
for the degree of Doctor of Philosophy

School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom
September 2005

Abstract

This thesis investigates planning in multi agent environments. A hypothesis is formed that in order to function effectively in the “real world”, agents need to be able to plan *efficiently* and *independently* within a social context. Standard refinement planning algorithms are unsuited to such situations because each agent is essentially situated in a changing external world.

Most multi agent planning approaches avoid this problem by centralising some aspect of planning. This reduces the multi agent problem to a set of single agent problems, enabling the use of refinement planning techniques but compromising agents independence.

From an independence point of view it is preferable to decentralise planning, allowing agents to exchange relevant information as they plan. This approach has received little attention in the literature, arguably because of the inapplicability of refinement planning.

In this thesis, a novel decentralised “distributed local planning” algorithm is developed and compared to related centralised and plan merging approaches. While the novel approach is found to be slower than its counterparts, it is shown to be applicable to a wider range of problems than plan merging. The advantages and disadvantages of each approach are discussed, and research is suggested for further development of distributed planning techniques.

Acknowledgements

I would like to express my gratitude to my supervisor, Aaron Sloman, for his continued help and guidance, and for repeatedly showing me that problems in AI are larger, richer and far more complex than I had previously considered.

Many thanks also to Nick Hawes, Manfred Kerber and Mark Ryan, who have provided much valuable input.

Special mention must go to my house-mates, David Brooks and Will Byrne, with whom I have spent many valuable hours drinking inspirational cups of coffee, shouting at computer screens and discussing problems over the living room whiteboard.

I would also like to thank my friends and fellow research students at the University of Birmingham for their procrastinatory input and their sage (and sometimes constructive) advice. I won't mention names for fear of leaving someone out: you know who you are.

Finally, I would like to offer the greatest thanks to my Mum and Dad, whose faith, support and encouragement have been indispensable over the last few years.

Contents

1	Introduction	1
1.1	A taxonomy of multi agent planning problems	5
1.1.1	What is an agent?	5
1.1.2	Common properties of problems	8
1.1.3	Problem specific properties of goals and plans	10
1.1.4	Problem specific properties of agents and the environment	13
1.1.5	Problem specific properties of sensing and execution	16
1.2	Planning problems addressed in this thesis	18
1.2.1	Reasons to cooperate	19
1.2.2	Motivational examples	21
1.3	Summary of contributions	23
2	Review of the literature	26
2.1	Single agent planning	26
2.1.1	Refinement planning	27
2.1.2	STRIPS representation	32
2.1.3	State space planning	35
2.1.4	Least commitment planning	37

2.1.5	Hierarchical Task Network planning	39
2.1.6	Local search planning	43
2.2	Multi agent planning	46
2.2.1	Centralised planning	46
2.2.2	Plan merging	48
2.2.3	Distributed planning	53
2.3	Summary of multi agent planning approaches	58
3	A common planning mechanism	61
3.1	Requirements	62
3.1.1	Joint and multi executive plans	62
3.1.2	Expressive temporal model	63
3.1.3	Flexible refinement	64
3.1.4	Accurate heuristics	65
3.2	Overview of task trees and summary information	65
3.2.1	Task trees	66
3.2.2	Histories	68
3.2.3	Summary information	69
3.2.4	Heuristics and test functions	71
3.2.5	Advantages and limitations of the approach	72
3.3	Propositional plans	74
3.3.1	Agents and ownership	75
3.3.2	World state	75
3.3.3	Actions and tasks	77

3.3.4	Task networks and temporal constraints	78
3.3.5	Problem definitions: initial plans and methods	85
3.3.6	Task trees	86
3.3.7	Summary conditions	89
3.3.8	Achieving, clobbering and undoing	91
3.3.9	Summarising task tree nodes	96
3.4	Propositional planning	100
3.4.1	Abstract detection of solutions and failures	100
3.4.2	Planning operators	102
3.4.3	Plan refinements	105
3.4.4	Simplifying task trees	108
3.4.5	Splitting and merging plans	110
3.5	First order plans and planning	111
3.5.1	First order plans	111
3.5.2	Approaches to handling first order problems	115
3.5.3	Planning with first order task trees	118
3.6	Recursive problems	121
3.6.1	Estimating optimal tree size	122
3.6.2	Adaptive task tree resizing	127
3.7	Task tree generation algorithm	127
3.8	Summary of planning mechanisms	129
4	Planning algorithms	130
4.1	Client-server model	131

4.2	Centralised planning	132
4.3	Plan-then-merge	134
4.4	Distributed local planning	139
4.4.1	Distributed constraint satisfaction	143
4.4.2	Distributed planning with DisCSP techniques	146
4.5	Summary of planning algorithms	152
5	Experiments and empirical analysis	154
5.1	Evaluation criteria	154
5.2	Experimental domains	156
5.2.1	The Blocksworld domain	156
5.2.2	The Navigation domain	159
5.2.3	The Holes domain	161
5.3	Comparison of planning mechanisms	163
5.4	Comparison of planning approaches	166
5.4.1	Blocksworld problems	168
5.4.2	Navigation problems	175
5.4.3	Holes problems	183
5.5	Summary of empirical analysis	192
6	Conclusions and future work	194
6.1	Summary of contributions	194
6.1.1	Taxonomy of multi agent planning problems	194
6.1.2	Approaches to multi agent planning	195
6.1.3	Common planning mechanism for multi agent planning	196

6.1.4	Comparison of multi agent planning approaches	198
6.1.5	Conclusions	203
6.2	Future work	204
A	Guide to notation	207
A.1	Data structures	207
A.2	Pseudocode	208
B	Domain descriptions and sample problems	210
B.1	Blocksworld	210
B.1.1	Domain description	210
B.1.2	Sample problem: bwReverse_3	212
B.2	Navigation	213
B.2.1	Domain description	213
B.2.2	Sample problem: navRing_3_2	214
B.3	Holes	215
B.3.1	Domain description	215
B.3.2	Sample problem: holesSpecial_3_1_3_1	218
C	Experimental data	220

List of Figures

1.1	Sample Blocksworld problem.	21
1.2	The <i>airlock</i> problem.	22
1.3	Sample pegs-and-holes problem.	23
2.1	Generalised algorithm for refinement planning.	29
2.2	Refinement search in a space of candidate plans.	29
2.3	Planning graph for a simple route planning problem.	37
2.4	Example of HTN planning: preparing an exotic meal.	42
2.5	Local search in a space of candidate plans.	44
2.6	Flow of information in a simple centralised planning system.	46
2.7	Flow of information in a simple plan merging system.	48
2.8	Flow of information in a simple distributed local planning system.	55
2.9	A distributed goal search tree.	57
3.1	Example HTN planning problem: going to work/town.	66
3.2	Preconditions and effects of primitive tasks from Figure 3.1.	67
3.3	Task tree for the abstract task and methods in Figure 3.1.	68
3.4	Summary information for the task tree in Figure 3.3.	70
3.5	Example of a recursive task: the <i>clear</i> task from Blocksworld.	74

3.6	Time-line of a precondition state constraint.	76
3.7	Time-line of a postcondition state constraint.	77
3.8	Allen’s thirteen <i>basic temporal relationships</i> between two finite intervals.	81
3.9	Algorithm for the commutative temporal operator <code>inv</code>	82
3.10	Algorithm for the transitive temporal operator <code>trans</code>	83
3.11	Propositional task tree for the problem in Figure 3.1.	87
3.12	Algorithm for propositional task tree generation.	88
3.13	Algorithm for summarising a task network.	98
3.14	Algorithm for summarising a task.	99
3.15	Algorithm for, and example of, the propositional <code>select_op</code> planning operator.	103
3.16	Algorithm for, and example of, the propositional <code>block_op</code> planning operator.	103
3.17	Algorithm for, and example of, the propositional <code>decompose_op</code> planning operator.	104
3.18	Algorithm for the <code>order_op</code> and <code>add_env_op</code> planning operators.	105
3.19	Algorithm for the propositional <code>decompose_ref</code> refinement.	106
3.20	Temporal relationships created by <code>order_ref</code> when an achiever is selected.	107
3.21	Temporal relationships created by <code>order_ref</code> when no achiever is selected.	107
3.22	Algorithm for the propositional <code>order_ref</code> refinement.	108
3.23	Example of simplification.	109
3.24	Algorithm for simplifying a propositional task tree.	110
3.25	Example type hierarchy and planning variables.	112
3.26	Algorithm for “unification” in first order problems.	114
3.27	Initial plan and example method from a first order package delivery domain.	115
3.28	Algorithm for the first order <code>bind_op</code> planning operator.	119

3.29	Algorithm for the first order decompose_op planning operator.	119
3.30	Algorithm for the first order decompose_ref refinement.	120
3.31	Abbreviated methods for single robot navigation.	122
3.32	Single robot navigation problem.	123
3.33	Initial task tree for the problem in Figure 3.32.	123
3.34	Abbreviated methods for multi robot navigation.	124
3.35	Multi robot navigation problem demonstrating shortcomings of value counting.	125
3.36	Solutions to the problem in Figure 3.35.	125
3.37	Methods for the independent two robot navigation action, <i>travel_avoid</i>	126
3.38	The unit of recursion of a method <i>m</i>	126
3.39	Algorithm for first order and compiled propositional task tree generation.	128
4.1	Algorithm for centralised planning.	133
4.2	Two agent Blocksworld problem suitable for the plan-then-merge approach.	135
4.3	Solutions to the individual problems in Figure 4.2.	136
4.4	Coordinated joint plan created by merging the individual plans in Figure 4.3.	137
4.5	Rules for client-server communication in distributed local planning.	141
4.6	Example constraint satisfaction problem.	143
4.7	Algorithm for distributed local planning.	147
4.8	Algorithms for message passing in distributed local planning.	151
5.1	Redundancy in level 3 <i>achieve_clear</i> tasks in <i>bwReverse_4</i>	158
5.2	Topology of <i>navLine_5_1</i> to <i>navStar_5_4</i>	160
5.3	Topology of <i>navRing_6_1</i> to <i>navRing_6_4</i>	160
5.4	Topology of <i>navStar_3_y</i> to <i>navStar_6_y</i>	161

5.5	Planning and plan merging times for two agent and three agent plan-then-merge for <i>bwReverseSeq</i> problems.	169
5.6	Individual plans produced during three agent plan-then-merge for <i>bwReverseRobin_10</i> and <i>bwReverseSeq_10</i>	170
5.7	Average solution times for <i>bwRandom</i> problems.	173
5.8	Plot of heuristic value against time showing weight explosion in distributed local planning.	175
5.9	Comparison of solutions times for centralised planning and plan-then-merge for <i>navLine</i> problems.	177
5.10	Sub-optimal plan produced by centralised planning for <i>navRing_5_3</i>	179
5.11	Plot of heuristic value against time showing a heuristic plateau in distributed local planning.	182
5.12	Plot of heuristic value against time showing escape from a plateau in distributed local planning.	183
5.13	Example <i>holesGeneral</i> problem.	186
5.14	Initial task tree for the problem in Figure 5.13.	186
5.15	Algorithm for domain specific task tree simplification in Holes problems.	187
5.16	Average solution times for <i>holesGeneral</i> problems.	189
5.17	Average solution times for <i>holesSpecial</i> problems.	191

List of Tables

3.1	Inverses of members of the BTR.	83
3.2	Transitive relationships between members of the BTR.	84
5.1	Tree generation and planning data for <i>bwUnstack</i> and <i>navLine</i> problems.	164
5.2	Success rates and average solution times for <i>bwSwap</i> problems.	168
5.3	Average numbers of tasks per plan for <i>bwSwap</i> problems.	168
5.4	Success rates and average solution times for <i>bwReverse</i> problems.	171
5.5	Average numbers of tasks per plan for <i>bwReverse</i> problems.	171
5.6	Success rates and average solution times for <i>bwRandom</i> problems.	172
5.7	Average numbers of tasks per plan for <i>bwRandom</i> problems.	174
5.8	Success rates and average solution times for <i>navLine</i> problems.	176
5.9	Average numbers of tasks per plan for <i>navLine</i> problems.	178
5.10	Success rates and average solution times for <i>navRing</i> problems.	178
5.11	Average numbers of tasks per plan for <i>navRing</i> problems.	180
5.12	Success rates and average solution times for <i>navStar</i> problems.	181
5.13	Average numbers of tasks per plan for <i>navStar</i> problems.	184
5.14	Success rates and average solution times for <i>holesGeneral</i> problems.	188
5.15	Success rates and average planning times for <i>holesSpecial</i> problems.	191

C.1	Complete experimental data for <i>bwSwap</i> problems.	222
C.2	Complete experimental data for <i>bwReverseRobin</i> problems.	222
C.3	Complete experimental data for <i>bwRandom</i> problems.	222
C.4	Complete experimental data for <i>navLine</i> problems.	223
C.5	Complete experimental data for <i>navRing</i> problems.	223
C.6	Complete experimental data for <i>navStar</i> problems.	224
C.7	Complete experimental data for <i>holesGeneral</i> problems.	225
C.8	Complete experimental data for <i>holesSpecial</i> problems.	226

Chapter 1

Introduction

The concept of an agent is important in modern computer science, and particularly in *Artificial Intelligence (AI)*. There are many different interpretations of agency in the literature, one of the most general being that of Franklin and Graesser (1996):

“An *autonomous agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.”

Wooldridge and Jennings (1994) suggest a more specific definition that captures some features commonly expected of agents in qualifying adjectives:

- *Autonomous* agents operate without direct intervention from humans or others;
- *Proactive* agents follow their own goals and take the initiative to achieve them;
- *Reactive* agents perceive their environment and react to changes;
- *Social* agents interact with other agents in their environment.

In other words, an agent is a problem solving process that is capable of acting independently and continuously to achieve its own internal goals, but that is capable of responding to factors in its environment such as unpredictable changes to the state of the world and the presence of other agents.

This is clearly a desirable ideal in AI. An agent that embodies these concepts would be able to perform complex tasks, overcoming unexpected problems with no human intervention, sharing information and resources with other agents to save time and effort for all participants. Goals could be directly set by a human user or could arise in response to environmental factors such as the availability of new information or resources.

Planning Agents typically have goals that cannot be achieved with a single action. In these cases it is useful for an agent to build a *plan* (or a number of plans) for what to do. In its simplest form a plan is a sequence of *actions* to perform, although much more complex models of time, action and resource usage are also possible. The field of *AI planning* has long been concerned with building systems that plan proactively and autonomously. However, many planning algorithms make assumptions that are incompatible with the reactive and social aspects of agency. Two assumptions follow that are relevant to this discussion:

1. The world can only change as a result of the actions in the agent's plan.
2. The planning agent is aware of the entire state of the world.

Planning and agency In order to plan in “real world” situations, agents need to be able to respond to external factors such as the arrival of new goals, unexpected changes in the environment or the presence of other agents. Such factors may cause goals or plans to become invalid during planning or plan execution. This may necessitate the changing of goals, or the recreation or “repair” of current plans. In addition, because changes may be occurring continuously, planning and plan execution must be treated as interleaving processes rather than a cycle of “create plan” and “execute plan” steps.

Multi agent planning Both of the assumptions above are violated in some way in many multi agent situations. No agent has exclusive control over the state of world because each agent is independently forming its own plan to achieve its own goals. In addition, agents cannot be aware of the entire state of the world because each agent has internal state (goals, candidate plans and so on) that cannot be directly accessed by other agents¹.

¹If agents are able to directly sense each others' internal state, then the second assumption may not be violated.

There are four ways of dealing with these violations, any or all of which can be used in different combinations or sequences to produce a particular approach to multi agent planning:

Isolated planning Agents can ignore each other, producing plans in isolation and *hoping* they will be executable or useful in later stages of planning.

Centralised planning Agents can outsource planning to a single agent, which is capable of producing plans for everybody without worrying about the implications of multi agent search.

Plan merging Agents can plan in isolation and employ a third party to take the completed plans and coordinate them to remove conflicts.

Distributed planning Agents can plan socially, communicating with each other as they progress to make sure their plans are executable when they finish.

The problem with isolated planning is that, if agents ignore each other during planning, conflicts may arise that render their plans useless during execution. For example, if Alice plans to use the car to go to town, her plan becomes invalid if Bob takes the car out to visit friends for the rest of the day. Centralised planning avoids this by giving one agent complete knowledge of the environment. However, centralisation is not without its disadvantages. If agents outsource planning or other problem solving activities, they reduce their own control over how plans are formed. This raises issues of other agents' interpretation (or misinterpretation) of their goals, knowledge and opinions, along with issues of privacy and trust if agents are in competition. Centralising planning can also create a single point of failure or a processing bottleneck in the team.

Plan merging and distributed planning are *decentralised* techniques that can be used when centralisation is inappropriate. Plan merging (Section 2.2.2) is the most popular current approach to multi agent planning: it essentially decomposes the multi agent problem into a number of smaller single agent problems, that are not allowed to interact with one another. The disadvantage of this approach is that the decomposition can prevent the agents finding solutions to some problems, when interaction is required during planning (Section 1.2.2). Distributed planning (Section 2.2.3) is ideal from an independence perspective, as agents can share as much or as

little information during planning as is necessary to find solution plans, without relying on a central process. However, because of the problems of changing external environments mentioned above, distributed planning cannot be implemented with “standard” planning techniques that make the assumptions listed on page 2.

Agent independence The term *independence* is used to refer to a measure of how much an agent relies (or does not rely) on other agents during planning. Agents have higher independence if they are able to plan without the direct assistance of others. For example, if Alice allows Bob to plan her day for her, she has little independence because she is relying on Bob to make all the decisions: she may well end up dropping Bob off at his friends’ house and picking him up again late at night, when really she would prefer some other arrangement. If, on the other hand, Alice and Bob are able to arrive at a mutual decision about car use, Alice has a say in the decision making process. She may persuade Bob to leave a little later to go out so she has time to go to town beforehand. The more independence agents have, the more they are able to adapt to different situations and problems, whether they are small or large, simple or complex, isolated or social.

Summary of this thesis Most current approaches to multi agent planning are based on centralised planning or plan merging. In both of these approaches, some agent independence is sacrificed to allow the use of single agent planning techniques, which are well developed in the literature. This thesis compares implementations of each with a new type of approach called *distributed local planning*. In this approach, agents plan concurrently and exchange coordination messages as they do so. This kind of distributed approach is needed if agent independence is to be preserved.

To ensure a fair comparison, the three approaches are implemented on top of a common HTN planning mechanism called the *Multiagent Planning Framework (MPF)*. MPF is an extension of work done by Clement (2002) in which *summary information* (Section 3.2) is extracted from the agents’ plans to allow them to reason about potential conflicts at high levels of abstraction. Clement’s work is restricted to propositional, non-recursive HTN problems (Section 2.1.5). Many interesting HTN problems contain first order state information and recursive methods;

MPP's extensions allow it to represent these kinds of problems.

The three approaches mentioned are compared empirically in Chapter 5 on a number of planning problems, including problems in which agents have heterogeneous planning abilities, resource control and knowledge of world state. Comparison criteria include: the approaches' efficiency (see below), the different types of problem they are able to solve, and the independence they offer agents. While distributed local planning is found to generally be the slowest approach of the three, it is able to solve some problems that plan merging cannot. The strengths and weaknesses in all three algorithms are discussed in Chapter 6, and a revised plan for implementing distributed local planning is outlined as future work that will avoid some of the weaknesses in the current implementation.

Unfortunately, the range of problems in multi agent planning is far too large to cover within the time and space available. The next section provides a taxonomy of problems and issues in multi agent planning, from which a specific subset is chosen in Section 1.2 for empirical analysis in Chapters 3 to 5. This chapter concludes with a summary of major contributions in Section 1.3.

1.1 A taxonomy of multi agent planning problems

The term “multi agent planning problem” can be used to describe a large and diverse range of AI problems. This section attempts to classify these problems by introducing a common problem definition and features that are common to specific problem subsets. Such a taxonomy is useful, firstly because it removes ambiguity in terminology used in the rest of the thesis, and secondly because it helps define the range of problems present in a particular scenario or solvable using a particular technique.

1.1.1 What is an agent?

The term “agent” is used with several different meanings in planning literature. Each meaning places a different interpretation on the phrase “multi agent planning”:

Agents as plan executives *Executives* are agents dedicated to the execution of plans. An exec-

utive is responsible for interpreting a plan and issuing appropriate instructions to motors, software interfaces, muscles and so on to make sure it is carried out successfully. Plans can be produced containing instructions for a single executive or multiple executives.

Single executive plans are either *coordinated* or *uncoordinated*. *Coordinated* plans contain enough information to guarantee success when they are executed in the presence of other executives; *uncoordinated* plans do not contain coordination information. Coordination information may include but is not limited to: timestamps on actions, prearranged points for inter-executive communication (Biggers and Ioerger, 2001), and diagnostic observations of world state (Weld et al., 1998).

Agents as subordinate problem solvers Specialist problem solvers may be used as components in a larger architecture such as a multi-process distributed planner or an architecture with more advanced cognitive abilities (Sloman, 2002). Planning is a complex task: problem solvers including specialist planning algorithms, scheduling algorithms, and domain specific knowledge bases may be applied to such diverse areas as action selection, resource allocation and causal reasoning. Homogeneous problem solvers may tackle large problems using divide-and-conquer techniques (Ephrati and Rosenschein, 1994), while heterogeneous solvers may be used to do special purpose reasoning or tackle a single subproblem with several algorithms (Wilkins and Myers, 1998).

Problem solvers can exist in master-slave relationships, in which one solver directly controls processes in the other, or peer-to-peer relationships, where they work independently and communicate only when necessary. An example of both types of arrangement is *RealPlan* (Srivastava et al., 2001), which is a planning system containing a *planner* that reasons about action orderings and a *scheduler* that allocates resources. *RealPlan-MS* is a master slave variant in which the planner periodically sends requests to the scheduler and backtracks if no resource allocation can be found. *RealPlan-PP*, however, is a peer-to-peer implementation in which the solvers search independently and use distributed search algorithms to agree on a solution.

Distribution can be used to decompose problems into “horizontal” subsumption layers (Brooks, 1991; Reynolds, 1999) as well as the “vertical” subproblems mentioned above. For example, a high level strategic planning agent may form travel plans at one level

and delegate reasoning about servo adjustments and motor speeds to lower level agents. Multi-layer architectures can be implemented using master-slave or peer-to-peer relationships.

Finally, planning may be just one process in a larger agent architecture with a wide range of capabilities. The space of possible architectures used to create “complete” agents is as large as the space of problems they are trying to solve (Sloman, 2000). This thesis, however, restricts itself to agents that just plan.

Agents as top level problem solvers Top level agents may be simple systems consisting of a single problem solver or larger architectures with a number of components. The difference between agents and problem solvers is that agents represent entire reasoning systems that are not contained within a larger system. This difference is one of a frame of reference: it is possible to think of agents as homunculi, lying inside each other, each existing in a different kind of local architecture or society.

In this thesis, the three concepts above are referred to as *executives*, *problem solvers* and *agents* respectively, although researchers in multi agent systems and multi agent planning have not adopted standard definitions of these terms. This terminology is suitable for representing simple systems but is insufficient for describing complex hierarchies of sensors, problem solvers, executives, motors and other components. The reader is referred to Franklin and Graesser (1996), Wooldridge and Jennings (1994) and Doran et al. (1997) for a more detailed discussion of what it is to be an agent.

Independence may or may not be important for problem solvers, depending on the architecture in which they operate. Independence is important for agents, however, as they may have to be able to deal with a range of different situations, including those in which they are isolated and those in which other agents are present.

Joint and individual plans The term *joint plan* is commonly used in the literature to mean “a plan for multiple agents”. Unfortunately, because of the ambiguity in the definition of “agent”, this term can have several meanings. The definition used in this thesis follows from the definition of “agent” above: joint plans are created by or for multiple planning agents. *This is in*

contrast with the most commonly accepted definition of a joint plan in the literature, which is a plan for multiple executives. In this thesis, plans involving multiple executives are simply referred to as *multi executive plans*. This research is concerned only with the creation of joint plans: the creation and execution of multi executive plans poses a very different set of problems and is beyond the scope of the thesis (examples include: Rosenschein, 1982; Biggers and Ioerger, 2001; Browning et al., 2004).

1.1.2 Common properties of problems

Multi agent planning problems, however simple or complex, have some properties in common. This section describes a set of properties upon which the definition of multi agent planning in this thesis is based. Subsequent sections discuss extensions to the problem that may arise in certain situations but not others.

Agents and the environment A *multi agent system* consists of a group of agents in a shared environment. Each agent has an *internal state* consisting of goals, beliefs, knowledge, plans and other such information. The environment also has state, referred to as *external state*, which agents can change by performing actions through the execution of their plans. Agents only have direct access to information in their own internal states. External state information must be obtained by *sensing* the environment. Other agents' internal state must either be obtained through *communication* or deduced from information gained by sensing the effects of agents' actions.

A *multi agent planning problem* consists of a multi agent system where each agent is capable of planning and has its own goals. Goals are achieved by executing plans, and plans have to be created before they are executed. There is always the risk that planned actions may become impossible to execute in the time between planning and execution. This may happen because of the actions of another agent or because of some unpredicted environmental event, such as a sudden change in weather or the unexpected breakdown of machinery.

Goals Goals form the basis of agents' pro-activeness, and are an essential part of agents' state. In the simplest planning problems goals are fixed, although in extended problems goals may be created or deleted by a number of factors (Section 1.1.3). At any time an agent has a subset of its goals selected for planning². Goals are specified as desired subsets of external state using some planning formalism. Depending on the expressiveness of the formalism, extra goal features such as temporal and resource requirements may also be specified (Section 1.1.3).

Plans and executives Plans are carried out by *executing* actions in the environment. Depending on the dynamics of goals and of the environment, execution may occur after planning, may be interleaved or concurrent with planning, or may simply be ignored.

In complex environments where plan execution is non trivial, actions can be modelled as capabilities of *executives* such as vehicles, manipulators, servos and motors. However, this model is not strictly necessary: executives may be overlooked in favour of more traditional representations of action (Section 2.1). The modelling of executives only becomes strictly necessary when executives can communicate back to the planning agent, or when agents can explicitly exchange control of executives³.

Sensing and communication Agents only have direct access to their own internal state. Strictly, external state information must be obtained through *sensing* (see below), although this can be ignored if sensors are assumed to have infinite range, responsiveness and reliability. Because agents cannot directly sense each others' state, they must exchange information about goals and plans by *communicating* with one another.

One situation where communication (and possibly sensing) may be ignored is where agents have shared internal storage. In this scenario the agents *may* have access to all of each others' internal information. Such a situation may occur, for example, when agents with different planning abilities are working on alternative solutions to the same problem (Melis and Meier, 2000).

²Goals are referred to as *desires* in *Belief Desire Intention (BDI)* nomenclature (Wooldridge and Jennings, 1994; Wooldridge, 2000)

³While the phrase "executive" is not in common use, the notion of a plan executing agent is common in the *Distributed Continual Planning (DCP)* literature (desJardins et al., 2000).

The basic problem described above may be complicated in a number of ways. Specific properties that may cause problems are categorised by type and discussed in Sections 1.1.3 to 1.1.5 below.

1.1.3 Problem specific properties of goals and plans

Agents form plans to achieve goals. However, “goal” and “plan” are very broad terms encompassing many concepts. The following properties of goals and plans may cause problems in particular:

Generation of goals In the simplest planning problems goals are fixed for the duration of planning. However, in more complex problems goals may be generated or become obsolete in response to actions, changes in the environment, or changes in higher level goals of the agent (Nilsson, 1994; Gordon and Logan, 2002). There may also be deadlines or time windows dictating when goals apply (Hawes, 2003), and agents may have to switch between goals during planning or plan execution.

Implicit and explicitly stated goals The term “goal” is usually used to refer to the *explicitly stated* goals that an agent planning to achieve. Examples include “getting home from work”, “filling up with petrol”. There are many types of explicit goals, some of which are described below. However, agents also have implicit goals that are not stated as part of the planning problem but affect the way they behave: the creation and successful execution of a plan are themselves implicit goals. Other implicit goals may be encoded in the algorithms, heuristics and implicit preferences given to the planner. For example, plans may be preferred if they are shorter or involve fewer steps, if they use fewer resources or increase the agent’s knowledge of the environment. Implicit goals are significant in multi agent problems because they can cause agents’ interpretations of plans to differ.

Types of goals Explicit goals may be long- or short-term, and may be conflicting or contradictory. Explicit goals may also exist in hierarchies or have complex structure that depends on other factors. Consider, for example, the goal “*Fill the car with enough petrol to go out at the*

weekend.”. The goal depends on properties of the car, the time until the weekend, the intended activity once the weekend arrives and so on.

Conflicts between goals Multi agent planning involves searching for plans that achieve the goals of all agents in the multi agent system. If some of these goals are conflicting, such a task may be impossible. Traditionally, AI planning is an all-or-nothing process: either the problem is solved or it is not. In the real world, however, it may be better to settle for a partial solution to a problem than to fail and produce no plans at all (Rosenschein and Zlotkin, 1994).

Different situations may demand different definitions of success or failure. For example, an agent may still consider a plan to be acceptable if it only achieves a subset of the agent’s goals. Agents may get different payoffs depending on how they go about achieving certain goals, and may have to reason about which goals to select if not all can be achieved. In multi agent systems, agents may have to negotiate over which goals to achieve and how to achieve them.

Conflicts can occur between explicit or implicit goals, and may be completely or partially resolvable with planning and/or negotiation. Consider the following examples:

1. people trying to book a tennis court for separate games at the same time;
2. players trying to maximise their winnings in a game of poker;
3. explorers managing their use of a limited shared water supply;
4. deliverymen trying to deliver the day’s packages and get off work early.

Example 1 represents a set of incompatible explicit goals for which there is no resolution: one person can book the tennis court but the others must go without a game. Examples 2 and 3 involve semi-compatible explicit goals in competitive and cooperative situations: agents have to reason about their personal gain versus the gains of other agents and the group as a whole. In the poker scenario the players are unlikely to care about the success of their competitors. In the explorers’ scenario, however, it does the team no good if one explorer grows too weak from under-drinking. Example 4 would most likely represent a conflict between implicit goals: plans that are less time consuming are usually implicitly considered better than lengthy plans. If the deliverymen can choose who takes which packages and/or who is allocated a delivery

vehicle, then goal selection and planning will potentially involve conflicts of interest as one man finishing earlier will mean another finishing later.

Partial solutions In single agent planning, if a problem cannot be solved in its entirety, it may be possible to achieve a subset of its goals to produce a satisfactory solution. Similarly in multi agent planning, if a joint problem is not solvable in its entirety, it may be possible to solve a subset of the agents' individual solutions as a best possible alternative. The usefulness of partial solutions depends on agents' implicit and explicit goals and whether goals and/or resources are shared or can be reallocated.

Types of planning problem There are many different types of planning problem and system described in the planning literature. Planning formalisms vary considerably in the kinds of goals, actions and resources they can represent, their treatment of time and uncertainty, and their treatment of plan execution. A discussion of all of the different possibilities is beyond the scope of this chapter. The basic planning problem is described in Section 2.1.2 and relevant extensions are discussed throughout Section 2.1. For a more complete treatment the reader is referred to the excellent book by Ghallab et al. (2004).

Planning in the space of multi agent systems Rosenschein and Zlotkin (1994) define three types of problem from the point of view of negotiation and cooperation in multi agent systems:

Task oriented problems involve sets of *non-conflicting* tasks, that can be successfully executed in any order. Agents try to minimise costs (to themselves or the team as a whole) by deciding on an allocation of tasks from a pool that need to be performed (as in example 4 above). Tasks are accepted by an agent at a certain cost, determined by the time or resources necessary to achieve them, but they cannot interfere with one another. In planning, tasks are equivalent to non-conflicting explicit goals⁴. In task oriented problems, planning is almost of secondary importance to goal allocation because inter-agent conflicts during planning are impossible.

⁴The terms *goal* and *task* are defined in a variety of ways in the literature. There is no universally accepted definition of either term.

State oriented problems introduce side effects to actions, allowing conflicts to occur between plans for achieving tasks. Planning becomes important during and after goal assignment because agents' plans can potentially interfere and conflict. The distribution of goals between agents can have a significant effect on the number of conflicts that are possible between agents' plans.

Worth oriented problems introduce a notion of “worth” attached to each visitable state of the environment. This greatly increase the complexity of the problems, as agents have to consider the individual value of each action in their plans. Some complex metric planning and scheduling domains touch on this kind of issue. If, for example, an agent has a limited amount of fuel, the “worth” of any particular world state could be described as a function involving the number of goals achieved, the remaining fuel level and the distance to the nearest petrol station. Similar complexity can be attributed to problems in which the time taken to create and execute plans is significant.

Most of the problems and systems discussed in the planning literature fit into the state oriented domain category above, although task and worth oriented problems are also present. A uniform measure of plan quality is usually adopted that is independent of world state except for the achievement of goals. Such quality may be measured in terms of the time or number of steps (“*makespan*”) required to execute a plan, the amount of resources (energy, fuel and so on) it consumes, and in some cases the time taken during planning (Hawes, 2003). Measures of quality are normally implemented as quantitative or qualitative *heuristics* used during planning, and form the basis of one type of implicit goal.

1.1.4 Problem specific properties of agents and the environment

The following are problem specific properties of the planning agents themselves and the environment they inhabit.

Altruistic, self-interested and untrustworthy agents *Altruistic* agents are concerned not only with their own planning success, but with the success of other agents in the environment.

The assumption of altruism gives agents common definitions of success, failure, cost and reward, and avoids many issues of differences of opinion and potential mistrust.

If agents are not completely altruistic then they must be partially *self interested*. Completely self interested agents are only concerned with the creation of their own plans, although self interested agents in general may have partial interest in group success. Because self interested agents do not have a consensus of opinion on the “best” courses of action, they may disagree on the worth of a potential plan or come into competition for shared resources. Some kind of negotiation according to an agreed protocol may be used to help overcome these problems.

Self interested agents may be trustworthy or untrustworthy. *Untrustworthy* agents may use misinformation or miscommunication to *cheat* other agents or otherwise gain a competitive advantage. This kind of competitive situation is the subject of a lot game theory research. Many algorithms and protocols have been contributed in the literature to limit or prevent cheating during agent interactions. The reader is referred to the excellent book by Rosenschein and Zlotkin (1994) for a grounding in this area.

Issues of self interestedness and mistrust become important when agents have conflicting or differing implicit or explicit goals (see above). Some sort of negotiation will be needed to agree on a suitable subset of goals if some goals are conflicting. Even if the set of explicit goals (Section 1.1.3) are compatible, agents may not agree on the best way of achieving them. In these situations some sort of consensus will be required, whether it is achieved through explicit negotiation or adherence to a common planning algorithm.

Heterogeneous and homogeneous agent abilities Agents may have heterogeneous planning or reasoning abilities. Some agents may be able to solve particular types of problem or have access to resources that other agents do not. If agents are capable of planning, they may have access to different algorithms, resources, executives, knowledge and/or libraries of pre-built plans or *methods* (Section 2.1.5).

If an agent cannot or does not want to solve its own planning problem completely, it may try to “contract” parts of the problem out to other agents. Contracting involves publishing a problem that needs solving, accepting offers from other agents, and choosing an appropriate contractor. This is usually done by voting, bidding, or some other sort of negotiation. For

example, the *Contract Net Protocol (CNP)* (Smith, 1980) is a popular algorithm in multi agent systems (Durfee and Lesser, 1991; Decker and Lesser, 1992; Wilkins and Myers, 1998) based on a proposal and a set of quotes from interested contractors.

Contracting does not necessarily sacrifice independence if an agent *chooses* to participate in it, rather than being *forced* to participate by some rigidly defined planning algorithm.

Static and dynamic environments Single and multi agent planning problems can broadly be classified into two types: those where the environment is *dynamic* and those where it is *static*. Dynamic environments involve unplanned spontaneous changes: in a multi agent sense this means events that occur without any agent initiating them. Static environments can only change as a result of agents' actions, although in multi agent problems the actions of other agents must still be considered.

Agent organisations In some scenarios agents may take part in *organisations*. These may be imposed by a human designer or may be self-organising depending on the task at hand. Pairs of agents may be in master-slave or peer-to-peer relationships. Slave agents in master-slave relationships are “invoked” by their masters to perform specific tasks, and return results directly. Master-slave and peer-to-peer relationships between agents are similar to those mentioned between problem solvers in Section 1.1.1: the difference is down to a choice of frame of reference.

Open and closed multi agent systems An *open* multi agent system involves agents that have no previous knowledge of the system architecture (Alberti et al., 2004). For example, navigation and exploration agents may be free to wander around an environment that is much larger than their maximum sensing and communication range. These agents are forced to form *ad hoc* teams as others enter and leave the local vicinity: they cannot rely on a predefined team structure to organise searching and mapping activities. Agents may also break down or suffer mechanical failure. The presence or absence of one agent may have an effect on other agents if commitments have been made to particular courses of action, or if agents are forced to rely on one another for certain abilities. Depending on the mechanism used to assign goals to agents, when an agent

leaves the system others may have to take over its goals or plans.

1.1.5 Problem specific properties of sensing and execution

Sensing and acting form the interface with agents and their environment. Sensing, while potentially complex in “real world” problems, is often greatly simplified or taken for granted in the planning literature. Similarly, plan execution in the real world is a complex problem that is further complicated by the presence of multiple agents and executives.

Partially effective sensors Strictly, agents receive information about the environment through the use of *sensors*. Sensors may have limited scope such as maximum range, restrictions on line-of-sight, and so on, and may have limited reliability, resulting in the retrieval of distorted or probabilistic information.

In some cases sensors may be directly modelled as being part of the environment. In these cases, sensors will have associated external state information, and their scope or reliability may be changed by performing actions such as moving or focusing attention (Sloman, 2001). For example, an agent monitoring a security system may be able to pan and zoom cameras and activate night vision modes to gain information on potential intruders.

While partially effective sensing is not a property of multi agent planning *per se*, it may be greatly complicated if agents have to share information to get a complete picture of the environment, or if sensors with associated actions are shared between agents.

Partially effective actions Like sensors, some actions may have limited scope or reliability. For example, a mobile platform with a manipulator arm may have to be located correctly before the arm can be used to grip objects. This kind of action dependency is common in planning problems.

In some cases actions may have uncertain or unpredictable outcomes. For example, once the manipulator above has gripped an object, there may be a chance that it will drop it again. This kind of problem may be further complicated if agents are dependent on one other to perform actions in order to satisfy preconditions for their own executives.

Partially effective communications Strictly, communications must be sent and received using special communications actions and sensors. Because of this, communications may also have limited scope or reliability. Reliability may be limited further by whether or not other agents can be trusted to provide factual information.

Partial knowledge of external state In some problems agents may have incomplete knowledge of the external environment. This is problematic for planning as missing knowledge may lead to missed opportunities or solutions.

In single agent planning two basic approaches exist for dealing with incomplete knowledge: either the agent goes about gathering the information necessary before it starts planning, or it inserts sensing actions and conditional branches into its plan to defer sensing until execution time (Golden and Weld, 1996; Weld et al., 1998). In multi agent problems agents may have different pieces knowledge that they can share or exchange for mutual benefit. If communication is impossible or unreliable, it may be possible to pick up some information by analysing other agents' actions and using *plan recognition* techniques (Chapter 24 of Ghallab et al., 2004).

Probabilistic or uncertain knowledge Knowledge may be incorrect or conflicting⁵. In some domains, knowledge about the environment is so poor that external state cannot be known for certain.

Uncertainty can arise if actions or sensors are unreliable, or if some part of world state is not observable with available sensors. In multi agent problems this can also come about if communication with other agents is unreliable, either because the communication channel is unreliable or because the agents themselves have unreliable information or cannot be trusted. This is a significant field of research in planning: readers are referred to Chapters 16 to 18 of Ghallab et al. (2004) for an introduction.

⁵For this reason, the term *belief* is preferred in some models of agency such as BDI (Wooldridge and Jennings, 1994).

1.2 Planning problems addressed in this thesis

As stated in the introduction above, approaches are needed for multi agent planning that preserve the independence of the agents involved. The taxonomy of problems and literature review in Section 1.1 and Chapter 2 discuss a wide variety of problems and approaches concerned with all areas of multi agent planning. However, this set of problems is too large to be tackled in its entirety in this thesis.

The empirical work in Chapters 3 to 5 concentrates on a very specific subset of the problems from Section 1.1 in which many of the extended problems above are ignored. A problem description is included below, with terms from Section 1.1 *italicised* for emphasis:

Problems consist of a *closed team* of agents occupying a *closed, static environment*. Each agent has a *static, non-conflicting set of goals* to plan for, and is assumed to be *self interested* but *trustworthy*. Algorithms are examined for finding a set of consistent *individual plans* that achieve all of the agents goals while respecting, wherever possible, agents' *independence*, including their privacy and autonomy.

In some problems agents are able to sense the complete environment, while in others they are restricted to specific information. Similarly, resources and actions are either available to all agents or are assigned exclusively to specific agents. The team as a whole is always aware of the complete external state, and always has control over all relevant resources. Agents are initially unaware of each others' goals and internal state, but may communicate information as is necessary to ensure success.

The environment is *static*: sensing and execution are ignored. However, communication must be explicit. Actions and communications are reliable and deterministic. Agents may not exchange goals, resources or executives, but may communicate knowledge to be used in planning.

The planning problems considered are *state oriented* with propositional or first order predicate state information. Goals are expressed as abstract tasks in a *Hierarchical Task Network (HTN)* formalism (Section 2.1.5). Agents may have different libraries of methods (predefined subplans) with which to plan.

Goals are assigned to agents by a human user and the agents plan until a solution is found or failure is signalled. The agent team succeeds by creating a valid set of coordinated individual plans satisfying all agents' goals. Partial solutions are considered failures.

While these problems are an extremely simple subset of possibilities from Section 1.1, it is hoped that insight gained from designing and comparing approaches for their solution will be useful in more advanced problems in future research.

1.2.1 Reasons to cooperate

In problems such as those described above there are many reasons for agents to cooperate and exchange information:

Coordination of shared resources It is often useful to control the access and consumption of shared resources in the environment. Consider, for example, a workshop in which several manufacturing agents are building machine parts. The agents may want to coordinate their use of resources such as tools, workbenches and raw materials to prevent conflicts.

Coordination of shared actions In the workshop there may be parts of the manufacturing process that require several agents to act simultaneously or in a choreographed sequence. If one agent needs to lift a heavy part onto a workbench, for example, he can ask another agent to help by lifting the other end of the part. Whereas some shared activities may be arranged *ad hoc* during execution, lengthy or complex activities may require planning in advance.

Removal of redundancy If two or more agents have the same action planned for a similar time, it may be possible for them to delegate the action to a single agent, performing other actions in parallel and saving time and resources overall.

Heterogeneous execution capabilities At some point during a plan, one agent may be in a position to schedule an action that other agents cannot. If agents are mutually dependent on each other to perform actions, the success of the team may depend on such actions. An example of this is shown in Scenario 2 in the next section.

Heterogeneous planning capabilities One agent may have exclusive knowledge that can be used to help another agent form its plan. For example, if one manufacturing agent has a good knowledge of some specialised manufacturing process, it can help an unskilled agent plan by showing him what equipment to use and in what order. An example of this is shown in Scenario 3 in the next section.

There are also reasons why independence may be desirable:

Privacy and trust An agent may not trust other agents to plan on its behalf. Even if agents are assumed to be trustworthy, implicit and explicit goals may be open to interpretation and another agent may not always get things right. Agents may also want to keep information private.

Efficiency Even if other agents can be trusted, the sharing of complete knowledge may be inefficient. This thesis uses two measures of efficiency:

- *speed*: the time in which a solution can be found (or the problem is found to have no solution);
- *memory efficiency*: the amount of memory used during solution.

Independence and efficiency interact in a problem dependent way. In some cases, providing agents with more independence will increase the efficiency of the team: in others it will not.

Heterogeneous capabilities If agents have different specialisations, it may be difficult to exchange enough data to allow a centralised planner to reason about all possible actions and plans. Consider, for example, the problem of a software engineer trying to design a piece of software by analysing users' requirements: neither party is likely to be able to do this effectively without the assistance of the other.

In practice there are many more reasons where autonomy is useful. Agents may be unable to communicate often or reliably or may have to make costly preparations or detours to do so. Processing time may also be at a premium, requiring fast, "rough and ready" local planning rather than all encompassing centralised planning. While this thesis does not deal with these issues directly, it aims to provide a foundation on which such extensions can be considered.

1.2.2 Motivational examples

Most current approaches to multi agent planning rely on some sort of central process: they either reduce to centralised planning for multiple executives or to some sort of plan merging approach. The examples presented in this section are intended to demonstrate some situations in which other modes of multi agent planning might be useful.

Each of these examples consists of several agents in a shared environment. While a central planner could potentially solve any of the problems, the focus of this research is on possible multi agent solutions: it is assumed that the agents involved are unwilling to give away control of their own planning problems. The intention is to develop approaches on these simple problems that might be applicable to more complex situations, such as those suggested in the introduction, in which centralisation really is inappropriate or impossible.

Scenario 1 *Two children are playing with building blocks in three colours. The teacher has told them each to build a tower of three or more blocks. Each child is certain of which colours it wants to use and in which order it wants them. Unfortunately, there are not enough blocks to go around: the children must work out a way to build a single tower containing both the desired patterns.*

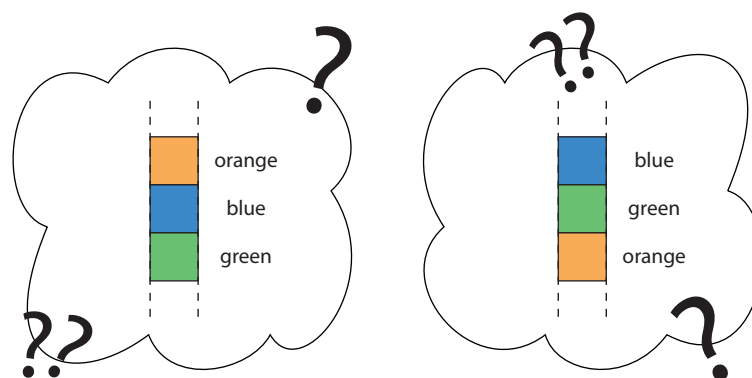


Figure 1.1: The Blocksworld problem from Scenario 1.

Each child has complete freedom to inspect and manipulate any block in the room. Assuming it is possible to build a tower that satisfies both of their designs, it is reasonable to assume that the children can come up with independent plans and coordinate them after planning. This kind of plan merging approach to multi agent planning is popular in the literature (Section 2.2.2).

Scenario 2 *Two robots wish to board a spaceship through an airlock. The airlock has two doors, only one of which is open at a time. The doors are operated by use of two buttons, one on the outside of the airlock and one on the inside. Each button toggles both doors when pressed, such that the one-open-one-closed state is always maintained.*

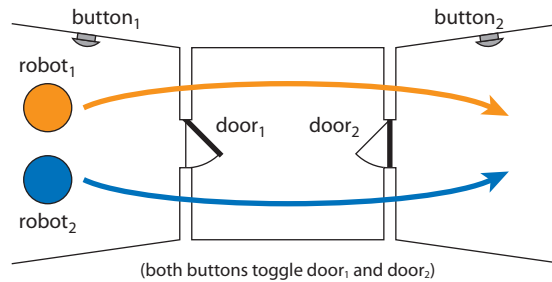


Figure 1.2: The *airlock problem* from Scenario 2.

This problem, referred to later as the “*airlock problem*”, is a specific instance of a general class of multi executive navigation domains in which the ability of one robot to move depends on the positions of other robots and environmental features.

Because there is no button on the inside of the airlock⁶, it is impossible for either robot to complete its own plan without the help of the other. Plan merging approaches are of no use, as without a set of uncoordinated individual plans, no merging can take place. Some sort of distributed planning paradigm is needed in which agents can exchange information *during* planning, if centralised planning is to be avoided.

Scenario 3 *Two robots are playing the children’s game in which pegs of different colours and shapes have to be placed in matching holes in a game board. Both robots are equipped with manipulator arms that are capable of grasping and moving the blocks. The robots have different sensors, however, which give them different information about the blocks. One robot has a high resolution black and white camera with which it can determine a block’s shape but not its colour. The other robot has a very low resolution colour camera with which it can determine colour but not shape.*

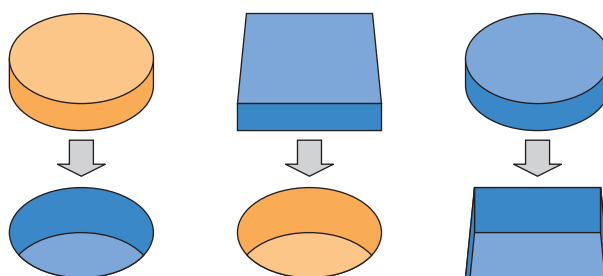


Figure 1.3: A pegs and holes problem from Scenario 3.

In this scenario it is unlikely that either robot will be able to independently come up with a plan that correctly places all of the blocks: they have to share information about which blocks are compatible with which holes. This can be done in a number of ways, one of which is as sets of resource constraints during planning. If one robot proposes a combination of peg and hole, the other robot can inspect the choice and veto it if it senses an incompatibility.

Most current approaches to multi agent planning involve the independent creation of individual plans for each agent and their subsequent merging to form a single joint plan. In Scenario 1 this is perfectly possible: each child can see and move all of the blocks in the room, and is probably quite capable of envisaging a plan that achieves all of its own goals without considering the actions of the other child. In Scenarios 2 and 3, agents have limited ability to change and sense the environment. This kind of problem often cannot be dealt with simply by planning and plan merging: other techniques are needed to ensure the agents can create their own plans (Section 2.2.3). One such approach, based on the exchange of summary information about resource usage, is presented in Section 4.4 of this thesis.

1.3 Summary of contributions

This thesis investigates various multi agent planning problems and approaches, providing a number of contributions to the field:

⁶Experts in spaceship design may recognise this as a design flaw.

Taxonomy of multi agent planning problems The taxonomy of problems in Section 1.1 clarifies some of the ambiguities present in the multi agent planning literature. The features mentioned should help identify the capabilities, strengths and weaknesses of existing and future research, together with possibilities for further advancement. The literature itself is reviewed in Chapter 2.

Multi agent planning approaches The literature review in Chapter 2 identifies a number of basic approaches to multi agent planning. Two of these approaches, *centralised planning* and *plan merging*, have been used extensively in the literature. A third approach, *distributed local planning*, has been used mainly in distributed constraint satisfaction (Yokoo and Hirayama, 2000). The three approaches are representative of the range of current approaches that may be suitable for solving the problems outlined in Section 1.2. Chapter 4 describes implementations of all three approaches. Novel techniques are used in the implementation of two approaches: *plan-then-merge* and *distributed local planning*.

Common planning mechanism for multi agent planning Unbiased comparison of the planning approaches requires them to be implemented in a standard way that eliminates as many implementational differences as possible. One part of this standardisation involves the implementation of a common planning mechanism on which the approaches can be built.

Clement and Durfee (1999a,b,c) developed a planning mechanism, *Concurrent Hierarchical Plans (CHiPs)*, for the centralised coordination of multiple agents' plans. The mechanism is capable of representing a restricted set of *Hierarchical Task Network (HTN)* planning problems. A number of novel extensions and alterations are made to CHiPs to produce the *Multi agent Planning Formalism (MPF)*, which is used as the basis of the empirical work in the thesis. Two versions of MPF are presented in Chapter 3:

1. A minor variant of CHiPs, called the *propositional Multi agent Planning Formalism (pMPF)*, is discussed in Sections 3.3 and 3.4. pMPF has a number of small extensions including annotations allowing it to be used with multiple agents. As the name suggests, actions and state information in pMPF are represented by Boolean propositions.

2. A number of extensions are made to pMPF to produce the *first order Multi agent Planning Formalism (fMPF)* in Sections 3.5 and 3.6. These extensions allow the representation of first order actions and state literals and a restricted set of *recursive* HTN problems.

pMPF and fMPF are compared empirically in Section 5.3, and their applicability and scalability to extended problems is discussed in Chapter 6.

Empirical comparison of approaches While none of the approaches identified is a clear favourite in all problems considered, each has its advantages in certain situations. An empirical and analytical comparison of the approaches is provided in Chapter 5. Chapter 6 discusses current shortcomings in the implementation of the algorithms and mechanisms presented, together with possibilities for future work, including the directions necessary for improving the independence of agents in multi agent planning problems.

The work presented below investigates, implements, analyses and evaluates potential approaches to multi agent planning in terms of their efficiency, the types of problem they can be applied to, and the independence they offer agents. Chapter 2 discusses relevant work on single and multi agent planning from the literature, Chapter 3 discusses the MPF planning mechanisms, and Chapter 4 discusses the implementation of centralised planning, plan merging, and distributed planning algorithms. Chapter 5 provides an empirical analysis of the performance of the algorithms applied to a number of “conventional” planning problems, and Chapter 6 reviews the contributions above in the light of the knowledge gained and presents a plan for future work.

Chapter 2

Review of the literature

Planning is an old and well established field. It is impossible to do it justice in its entirety here so this discussion will be restricted to material relevant to this thesis. Relevant research falls into two parts: *single agent planning* and *multi agent planning*. These parts are dealt with separately below.

Section 2.1 reviews current approaches to single agent planning. This is done for two reasons: to give the reader a basic understanding of the range of approaches, and to identify the approaches that are may be suitable for extension to multiple agents. Section 2.2 goes on to discuss relevant approaches to multi agent planning. Section 2.3 draws concluding remarks, and sets the scene for the development of a common mechanism (Chapter 3) and implementations of several algorithms for multi agent planning (Chapter 4).

2.1 Single agent planning

Planning is a process of *search*. The planning agent is given a set of input information such as the current state of the environment, a set of actions that may be performed and a set of goals to achieve, and searches through possible sequences of actions until it finds a suitable solution plan. This is a hard problem as the space of possible sequences can be very large and sometimes infinite. Successful planning approaches involve an informed systematic search through the possibilities until a solution plan is found, or until the agent is reasonably sure that

no solution exists (either the search space has been exhausted or a timeout interval has passed without a solution being found).

This section reviews current approaches to single agent planning in an attempt to identify approaches suitable for adaptation to multiple agents. Section 2.1.1 describes the *refinement planning paradigm*, which is not a specific approach but a framework in which many planning algorithms can be expressed and compared using common concepts and terminology. Refinement planning is used throughout the rest of this thesis in the discussion of single and multi agent planning mechanisms and algorithms alike. Sections 2.1.2 to 2.1.5 describe various refinement planning mechanisms that might be used as a basis for multi agent planning, and Section 2.1.6 describes alternative *local search* planning approaches that do not fit into the refinement framework.

A semi-formal notation is used to describe data structures and algorithms in several places in this chapter. A guide to the notation can be found in Appendix A.

2.1.1 Refinement planning

Given a specification for a problem, planning can be formulated as a search through possible *candidate action sequences* to find a solution that achieves the desired result. The refinement planning paradigm (Kambhampati, 1997) involves starting with a state representing the set of all possible action sequences and gradually shrinking it to a single solution sequence. Refinement planning maintains an *open list* of sets of candidates that have yet to be considered (see below), allowing the agent to perform a sound, complete, systematic search of “candidate space”, but limiting its applicability to environments that are static during planning. By contrast, *local search planning* (Section 2.1.6) does not provide soundness and completeness guarantees, but is more suited to dynamic environments.

Partial plans Sets of candidate action sequences are specified using sets of constraints called *partial plans*¹. A partial plan simultaneously represents all the action sequences that are consistent with its constraints. As constraints are added to the partial plan the set of candidate

¹Constraints are implicitly present in all plan representations: they do not need to be explicitly stated as part of a plan.

sequences decreases in size. The planner is finished when it has a partial plan in which all candidates are solutions.

Consider, for example, a planning formalism in which actions may be inserted at any point into a totally ordered list. The empty list represents all possible candidate sequences as no commitment has been made to any actions or orderings. The list $(go_to_shops, go_to_work)$ represents all candidate sequences in which the shops and work are visited, in that order but not necessarily immediately after one another. If the planning algorithm only allowed the addition of actions to the beginning or end of the list, the same list would represent all candidate sequences in which the shops and work are visited consecutively. Thus, the set of sequences represented by a partial plan is dependent on the plan representation used and the set of planning operators available to the agent.

In practice, depending on the partial plan representation chosen, it may not always be possible to represent the desired set of candidates at a particular stage of refinement using a single partial plan. For example, the list based representation from the example above is only capable of representing totally ordered action sequences, so it cannot be used to represent the plan “visit the shops before or after work”. Strictly speaking, refinement planning operates on *sets* of partial plans that represent all necessary candidates. For simplicity, however, this thesis assumes that a single partial plan is always sufficiently expressive to represent all candidates.

Refinement planning algorithm A generalised deterministic refinement planning algorithm is shown in Figure 2.1 and graphically in Figure 2.2. The algorithm takes an *initial* partial plan, representing a large set of candidate sequences, as input, and progressively refines it to produce a *solution* plan in which only solution sequences remain. The solution plan in Figure 2.2 contains a single solution sequence, but in practice several solutions can be represented in a single plan. At each iteration, the planning agent chooses a feature of the plan to improve, called a *refinement* (line 10). A refinement can be many things including the removal of a flaw, the extension of the plan by an action, or the making of a specific decision.

There may be more than one way of performing a given refinement. For example, there may

```

1  function bfs(initial) → solution:
2      create empty open_list
3      (open_list ← push(initial, open_list))
4      while ¬empty(open_list):
5          (plan, open_list) ← pop(open_list)
6          if is_solution(plan):
7              solution ← plan
8              return
9          else:
10             ref ← pick_refinement(plan) (not a backtracking point)
11             open_list ← push(plans(ref), open_list)
12     solution ← failure

```

Figure 2.1: Generalised algorithm for refinement planning.

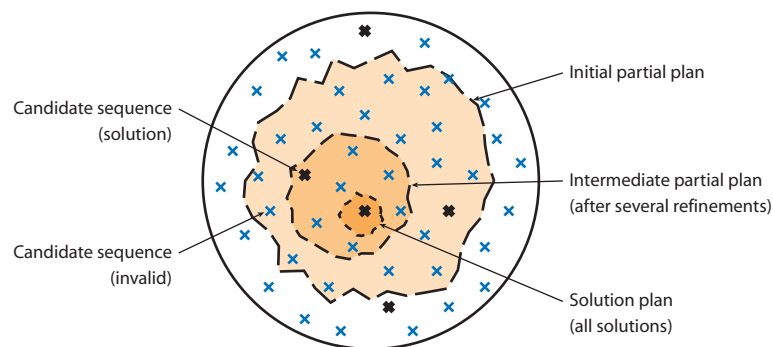


Figure 2.2: Refinement search in a space of candidate plans (adapted from Narayek, 2002).

be more than one way of resolving a particular flaw or performing a particular task. Each of these ways is embodied in a *planning operator* (or sequence of operators) leading to a valid partial plan. The agent chooses an appropriate operator and applies it to the plan (line 11). Each successive operator fills in more constraints, and the process continues until a solution plan is found (line 6) or no more valid plans can be generated.

Consider, for example, an agent trying to produce a list of up to five letters, chosen from the set $\{A, B, C, D, E\}$, representing any dictionary word². The complete set of candidate sequences can be generated by appending different combinations of letters to the end of an empty list.

²While lists of letters are not strictly the same thing as sequences of actions, they are analogous to the list based plans described above.

The agent starts with an empty letter sequence and adds it to its open list (line 3). At each iteration the agent removes a partial sequence from the list and checks if it is a dictionary word (line 6). If so, the agent returns the word as a solution. Otherwise, the agent applies the *append_letter* refinement (line 10), generating all possible sequences resulting from appending a single letter and adding all sequences of five letters or less back to the open list (line 11).

In most planning algorithms an agent will be able to perform more than one type of refinement. In the example above the agent could have been given a *prepend_letter* refinement as well as an *append_letter* one. A single refinement r_0 is picked per partial plan p_0 visited during search³. When r_0 is applied to p_0 , it should output a set of plans P_1 that, between them, represent the complete set of candidate sequences of p_0 . If each of the members of P_1 contains some kind of definite flaw, the agent can signal failure immediately: it does not have to backtrack and try an alternative refinement of p_0 . For example, there are no words in the English language that contain three or more identical consecutive vowels: if the agent in the example above were to come across the partial sequence “AAA”, it could discard it immediately without considering the derivatives “AAAA”, “AAAB”, “AAAC” and so on.

Planning approaches The refinement planning algorithm can be altered to resemble almost any modern planning algorithm by changing one or more of the following features, referred to collectively in this thesis as a *planning approach*:

Plan representation This is the type of data structure used to represent the plan, including features such as temporal and resource models. The choice of representation affects the sets of candidates that can be stored in a single partial plan. This affects how many plans the agent has to produce to cover all possible ways of performing a refinement, and consequently the number of plans that need to be visited per refinement in the worst case.

Refinements and operators The refinements available to the planner affect the efficiency, completeness and branching properties⁴ of the algorithm. They affect the ability of the

³Hence the word “picked” and not “chosen”, which has extra connotations in search.

⁴Branching properties include the branching factor, the redundancy of search states, the order in which candidates can be considered and so on.

planner to directly tackle specific flaws in a plan, minimise the production of redundant plans, and so on.

Choice of refinement The choice of refinement (`pick_refinement` function, line 10) has a large effect on the performance of the algorithm. Strategies such as least commitment (Weld, 1994) and the identification of separable subproblems (Korf, 1987) require the agent to identify refinements early on in planning that shape search later on. In particular, refinement choice affects:

- how quickly sets of invalid action sequences can be pruned from the search space;
- the ability of an agent to commit to plan features in a multi agent environment, for the benefit of fellow agents;
- the redundancy in plans visited during planning;
- whether second or third parties need to be contracted to help resolve conflicts in multi agent problems.

Heuristics Heuristics may be used to help pick the refinements that will produce the maximum benefit later on, and to choose the order in which to investigate partial plans once they have been generated.

Solution and failure test functions The `is_solution` function (line 6) is used to identify complete or partial solutions when they arise. In some cases this can be a complex problem, as it will not always be obvious whether some or all of the candidate action sequences in a partial plan are solutions. Other functions may also be made available to identify unresolvable flaws in some partial plans and prune the corresponding branches of search (as with the three vowel example above).

Search algorithm The `push` and `pop` functions can be implemented in various ways to produce different search algorithms. Stack like functions produce depth first search, queue like functions produce breadth first search and so on. The most common forms of search in planning are A* and best first (heuristic only A*) search, which treat the open list as a priority queue sorted by a heuristic measure of plan quality. Depending on the nature of the planning problem, however, other algorithms may prove more useful. Search algorithms are discussed in more detail by Russell and Norvig (1995, Chapter 3).

The aspects of planning listed above are collectively referred to in this thesis as a *planning approach*. The same aspects minus the planning algorithm are collectively referred to as a *planning mechanism*. Thus:

$$\textit{approach} = \textit{mechanism} + \textit{algorithm}$$

While this thesis is concerned with multi agent planning rather than single agent planning, a limited discussion of the basic approaches to single agent planning is required. Refinement based planning approaches can be divided into two categories: those that search in the space of *states* of the environment that may be visited by performing actions, and those that search in the space of possible *plans*. These paradigms, called *state space* and *plan space* planning respectively, are briefly introduced in the following sections. Section 2.1.2 introduces the *STRIPS action representation* that is common to many planning algorithms. Section 2.1.3 discusses state space approaches. Sections 2.1.4 and 2.1.5 discuss the two basic types of plan space planners: *least commitment* and *hierarchical* planners. Finally, Section 2.1.6 discusses a class of *local search planners* that do not fit into the refinement planning framework.

2.1.2 STRIPS representation

The first planner was the *General Problem Solver (GPS)* of Newell and Simon (1969). GPS could solve various problems, including planning and theorem proving, through the iterative application of logical rules. Work on planners has since focused on more specific formalisms better suited to the production of efficient planning algorithms⁵.

GPS was succeeded by *STRIPS* (Fikes and Nilsson, 1971), the most commonly known planning system. The major contribution of STRIPS was a compact action representation that is still used by many current planners. A STRIPS action consists of three parts: a *description*, a *precondition* formula and a set of *effects*⁶. For example the action of travelling between neighbouring towns

⁵In terms of the time and memory required to solve problems.

⁶Because of the close mapping between the actions and search operators in state space search, STRIPS actions are sometimes referred to as *STRIPS operators*. The former term is used here as search operators are different in different forms of planning.

might be represented as follows:

$$\begin{aligned} \text{action: } & \text{travel}(?a, ?b) \\ \text{preconditions: } & \text{at}(?a) \wedge \text{road}(?a, ?b) \\ \text{effects: } & \{ \text{at}(?b), \neg \text{at}(?a) \} \end{aligned}$$

where *travel* is the name of the action, *?a* and *?b* are variable arguments representing towns⁷ and *at(?x)* and *road(?x, ?y)* are Boolean valued state literals. In English this translates as:

Preconditions: To travel from *?a* to *?b* there must be a road from *?a* to *?b* and the traveller must be at *?a*.

Effects: Once the traveller has moved, he is no longer at *?a*: he is at *?b* instead.

Planning variables Variables are used to specify actions that can be applied to more than one object. They allow a partial specification of the arguments of an action. Actions and literals are referred to as *grounded* if all their arguments refer to objects in the world and *lifted* if they contain one or more variables. Variables are bound to world objects when an agent decides to commit to a specific action during planning.

Consider, for example, the list $(\text{travel}(\text{Birmingham}, ?x), \text{travel}(?x, \text{London}))$, representing a plan to travel from Birmingham to London via a third as yet unchosen city *?x*. Suppose there are three possible values of *?x*: *Farnham*, *Reading* and *Bristol*. *?x* is existentially quantified: it represents exactly *one* of the three possible values. Until *?x* is bound to one of the three values, the plan represents a disjunction of three sets of candidate plans: those involving travelling via Farnham, travelling via Reading, and travelling via Bristol. Once *?x* has been bound to a value, the plan takes on a more specific meaning and no longer represents the sets of candidates involving the other two locations.

Variables in preconditions are existentially quantified, referring to a single object in the world. Variables in effects must appear in the action description or preconditions. This allows the

⁷The convention in this thesis is to prefix all variable names with question marks.

planning agent to analyse the predicted state of the world at the time when the action is due to start and bind variables accordingly to produce the desired effects.

Plans A complete STRIPS plan is a totally ordered sequence of grounded actions. The following conditions hold for a plan:

- The plan is *valid* if the preconditions of each action match subsets of the world state immediately before the action is executed (taking into account the effects of previous actions).
- The plan is a *solution* to a problem if goals of the problem match a subset of the world state immediately after the last action in the plan is executed.

Extensions to STRIPS representation The STRIPS action and plan representation are extremely simple. They lack many features that are considered convenient or necessary to represent “real world” applications. Examples include:

1. universal quantification of variables in preconditions or effects;
2. “conditional action effects” that depend on world state at the time of execution;
3. decomposition of actions into smaller parts, or aggregation of actions into “macro actions” (for example for learning);
4. negative or disjunctive preconditions;
5. typed planning variables and world objects⁸;
6. actions with duration and/or delayed effects;
7. a quantitative temporal model;
8. metric or non-Boolean world state;
9. actions with uncertain or unpredictable effects;
10. sensing actions that reveal world state;
11. conditional branching depending on world state or action effects;
12. looping.

⁸While typing can be emulated with state literals such as *location(?x)*, the explicit use of types can be a more efficient approach (Section 3.5).

Some of these features are a convenience, reducing the work needed to encode planning problems but not actually adding any representational power. Other features represent types of problem and plan that simply cannot be represented using the STRIPS formalism.

Universal quantification and conditional action effects were first addressed by the *Action Description Language (ADL)* (Pednault, 1987) and later by the well known *Planning Domain Definition Language (PDDL)* (McDermott, 1998; Fox and Long, 2003). PDDL, in its various incarnations, has also provided support for many of the other features above, although noticeably not looping.

While many of the features above are useful for solving real world problems, only a few are strictly necessary for dealing with multiple agents. The most important features in this regard are a sufficiently expressive model of interactions and relationships between concurrent actions (Brenner, 2003). Early work by Georgeff (1983) on *process models* helped lay the groundwork for understanding in this area. Support for sensing actions or uncertainty are also potentially useful if agents have severely limited knowledge of the world, although less so if agents are able to exchange information before plan execution to get a complete picture of world state. Other features such as universal quantification and conditional effects reduce potential maintenance on plans in dynamic worlds where external changes may cause action effects to change.

2.1.3 State space planning

The first category of refinement planners, *state space planners*, include *STRIPS* itself. State space planning involves searching in the space of possible world states. Reachable states are determined by simulating the effects of actions. State space planners are given three initial inputs:

1. A set of symbols O representing objects in the world and a complete description of the initial world state I . World state is specified as a set of literals defined on tuples of elements from O .
2. The goal state G , also specified as a set of literals. G does not have to be a *complete* description of world state; unspecified literals are assumed to be unimportant and are

ignored when checking for goal states.

3. A set of actions A , in STRIPS or some equivalent representation.

They proceed in one of two ways:

Forward chaining planners start at the initial world state and search forward in time. Actions are added to the end of the plan, which is assumed to be totally ordered, until a state is reached that matches the goal state.

Regression planners do a similar search starting at the goal state and working backward towards the initial conditions.

Despite the intuitiveness of state space planning, the space of reachable world states is vast and for many years no good techniques were known for guiding algorithms to solutions (Chapter 4 of Ghallab et al., 2004). This meant that for a long time state space planners were only able to handle small problems and *plan space* planning (Sections 2.1.4 and 2.1.5) dominated. Recent work on *planning graphs* (Blum and Furst, 1997) has revived state space planning by providing a compact way of simultaneously representing alternative orderings of actions.

Planning graphs Planning graphs are a compact plan representation that form the basis of some of the fastest current planning techniques. A planning graph is a relaxed representation of possible action sequences and reachable world states. It consists of interleaving layers of propositions representing possible future states and possible choices of actions. Layers 1 and 2 represent possible state and action choices at time 1, layers 3 and 4 represent possible state and action choices at time 2 and so on (Figure 2.3).

The graph is constructed with a forward chaining search, each iteration extending it by one unit of time (or two layers). Because layers represent *possible* future states, individual layers can contain nodes that would be mutually exclusive in a single possible world. *Mutex arcs* are used to denote pairs of mutually exclusive state and action nodes in each layer. At each iteration the graph is searched for possible plan sequences if the goal state is a non-mutex subset of the final state layer. Plan retrieval is done with a backward search that makes heavy use of mutex information in the planning graph.

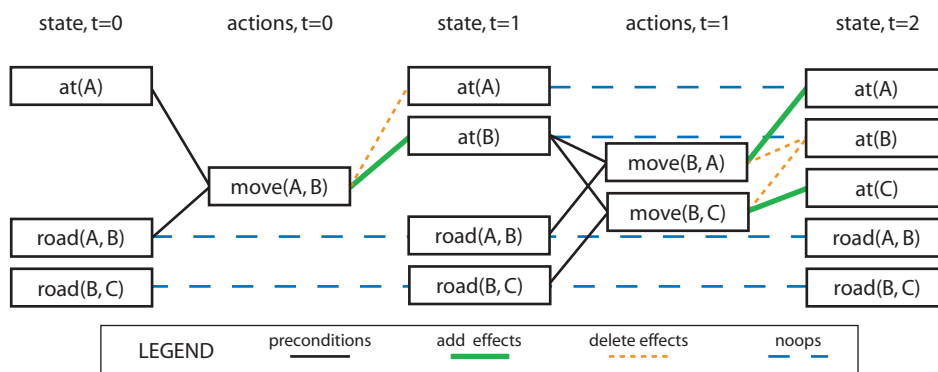


Figure 2.3: Planning graph for a simple route planning problem.

More recent extensions to planning graphs handle many extra features, including universal quantification and conditional effects, exploitation of symmetry, automatic type identification and exploitation, uncertainty and sensing actions (Weld, 1999), metric resources, quantitative time and durative actions with delayed effects (Do and Kambhampati, 2001). Many of the fastest modern planners (Bacchus and Kabanza, 2000; Hoffmann and Nebel, 2001; Smith and Weld, 1999) plan in state space and use planning graphs either directly or as the basis of heuristics.

2.1.4 Least commitment planning

Least commitment planning is one of two types of plan space refinement planning⁹ discussed in this thesis. It involves deferring decisions about the temporal orderings and variable bindings until they are required to resolve conflicts in the partial plan. Least commitment allows the planner to be flexible about parts of its plan until it has enough information to work out the best possible course of action. While many of the fastest current planning algorithms are based on state space planning, it is this flexibility that makes plan space planning, including least commitment planning, interesting from a multi agent perspective. Least commitment planners are often also referred to as “partial order planners” or “Partial Order Causal Link (POCL) planners”¹⁰. The following is a simplified version of the description of UCPOP provided by

⁹The *principle of least commitment*, introduced by Marr (1982), has been used in many fields of AI, including computer vision, planning and theorem proving.

¹⁰Technically HTN planners (Section 2.1.5) are also least commitment planners. Unfortunately there is no clear terminology that categorises all techniques with no overlap between categories.

Penberthy and Weld (1992):

A partial plan p is a tuple $\langle A, T, B, L \rangle$ where A is a set of actions, T is a set of temporal constraints, B is a set of variable bindings and L is a set of *causal links*. The preconditions and effects of an action $a \in A$ are referred to as $\text{pre}(a)$ and $\text{eff}(a)$ respectively. Actions are unordered and lifted by default; constraints are added to T and B where necessary to impose temporal orderings, ground variables and remove conflicts. By changing the language of implementation of T , a number of qualitative and quantitative temporal models can be implemented with varying complexity and expressiveness (Schwalb and Vila, 1998; Dechter et al., 1991).

A causal link is a structure $a_i \xrightarrow{l} a_j$ where a_i and a_j are actions in A , a_i is ordered before a_j and l is an effect of a_i and a precondition of a_j . a_i and a_j are referred to as the *provider* and *consumer* of the literal respectively. Causal links are added to the plan to *protect* preconditions, making sure they do not come under *threat* from actions with conflicting effects.

Least commitment planners are given two initial inputs:

1. An *initial plan* p where:

- A_p contains two dummy actions: a_0 and a_∞ ,
- $\text{eff}(a_0)$ represents the initial state of the world,
- $\text{pre}(a_\infty)$ represents the goal state,
- a_0 is ordered before a_∞ ,
- B_p is empty
- L_p is empty

2. A set of actions in STRIPS representation or some equivalent

They proceed by iteratively selecting an action a_j and unachieved precondition $l \in \text{pre}(a_j)$ and trying to achieve it by making the following refinements to the plan:

- adding a causal link $a_i \xrightarrow{l} a_j$ from an appropriate existing action a_i ;
- adding a new action a_i that may be used as a provider and a causal link $a_i \xrightarrow{l} a_j$;
- removing a threat to an existing causal link $a_i \xrightarrow{l} a_j$ from an action a_k with $\neg l \in \text{eff}(a_k)$ by adding appropriate constraints to T and/or B .

A valid solution is a plan with no unachieved preconditions and no threats to causal links.

Penberthy and Weld (1992) built on early least commitment planners such as *SNLP* (McAllester and Rosenblitt, 1991) and *TWEAK* (Chapman, 1987) to produce *UCPOP*, the first least commitment planner to handle ADL (conditional action effects and universal quantification). Since then they have gone on to produce *Zeno* (Penberthy and Weld, 1994), one of the most expressive current plan space planners, which is capable of handling domain features such as metric resources and durative actions with delayed and continuous effects.

2.1.5 Hierarchical Task Network planning

The earliest plan space planners (Sacerdoti, 1975; Tate, 1977) did not search in the space of task orderings and causal links, but rather in the space of *decompositions* of actions. These early planners gave rise to another form of plan space refinement planning: *Hierarchical Task Network (HTN)* planning. Like least commitment planning, hierarchical planning allows a certain amount of flexibility during planning. It also allows agents to reason at varying levels of abstraction, which could be useful for reducing communications overheads and limiting the complexity of inter-agent coordination in multi agent planning.

HTN planners are similar to least commitment planners in that they search in the space of partial plans and explicitly model temporal and variable binding constraints. However, they are different in the *way* they search the space: by considering the possible *decompositions* of *abstract tasks*. Just as the “flat” planners above are based on the notions of *actions*, *goals* and *plans*, HTN planners are based on notions of *tasks*, *task networks* and *methods*:

Tasks subsume actions and goals. Rather than planning to achieve a set of goals, an agent plans to perform a set of tasks. There are two types of task: *primitive* tasks that can be executed directly and *abstract* tasks that must be *decomposed* into smaller tasks during planning.

Task networks are used to represent plans and subplans. A task network n is a tuple $\langle A, T, B \rangle$ where A is a set of tasks, T is a set of temporal constraints on the members of A and B is a set of variable binding constraints.

Methods represent ways of decomposing abstract tasks. A method m is a tuple $\langle h, b \rangle$

where h is an abstract task and b is a task network representing a subplan that achieves the desired effects of h .

HTN planners are given two initial inputs:

1. A non-empty initial plan p containing abstract and primitive tasks that need to be performed.
2. A set of *methods* M providing ways of decomposing abstract tasks.

They proceed by iteratively selecting a task $t \in A_p$ and decomposing it using an applicable method $m \in M$ by substituting for t with b_m .

Primitive tasks have preconditions and effects in the same way STRIPS actions do. Abstract tasks have no preconditions and effects *per se*, although new preconditions and effects may be introduced when an abstract task is decomposed. A single abstract task may have more than one applicable method, so relevant preconditions and effects are not always known in advance. Conflicts introduced during decomposition are resolved as they are in least commitment planning by adding temporal or variable binding constraints¹¹.

Most HTN planners search only in the space of task decompositions¹². This is limiting because every combination of actions has to have been thought out in advance by the human designer of the method library. This is the major objection raised against HTN planning by supporters of “flat” planning techniques, but it does not prevent HTN planners being the most widely used planners in industry for two reasons:

1. HTN is an intuitive way of thinking about planning, and many planning systems in industry are *mixed initiative* systems that interact with human operators as well as performing automated search.
2. Human designers can encode problem specific knowledge into methods such that the planner produces predictable plans in a timely fashion.

¹¹HTN planners may or may not also explicitly represent causal links between tasks.

¹²A notable exception to this is *HyHTN* (McCluskey et al., 2002), which is a hybrid planning system capable of a combination of decomposition and state advancing search.

Consider, for example, a manufacturing scenario in which it is necessary for a particular machine to produce a diagnostic report every time it is used. The specification of this constraint in a “flat” planning formalism would require the addition of state information and action preconditions and effects to ensure that every time a *use_machine* action is added to a plan, a *produce_report* action is also added. This is a non-trivial adjustment to the planning domain, and would add an extra layer of complexity to the problems given to the planner. In HTN planning, however, the *produce_report* task would simply need to be added to the relevant method, and the planner would not need to do any extra processing to ensure its use at the relevant points in the plan.

Recursive versus non-recursive domains HTN planning problems can broadly be classified into ones that contain *recursive methods* and ones that are not. A method m is recursive if B_m contains a subtask, some part of which may be decomposed with m . Blocksworld is a classic example of a recursive HTN domain:

In order to pick up block A I have to make sure there is nothing on top of it.
If there is a block B on top of A I must pick it up and place it on the table.
In order to pick up block B I have to make sure there is nothing on top of it.
If there is a block C on top of B I must pick it up and place it on the table...

The issue of recursive and non-recursive problems is revisited in more detail in Chapter 3.

Conflict detection HTN planners are prone to backtracking because preconditions and effects are only added to the plan after tasks have been decomposed.

Consider, for example, an agent planning the preparation of an exotic meal as shown in Figure 2.4. This is done in three decompositions: first, the agent chooses whether to prepare a Chinese or an Indian meal, then it shops for the ingredients, and finally it cooks and eats the meal. Chinese meals involve eating with chopsticks: if the agent doesn't have any chopsticks, it will be unable to find a suitable

plan if it uses the *prepare_chinese* method¹³. A human would immediately recognise this and choose to make an Indian meal. An uninformed HTN agent, however, may choose *prepare_chinese* and make all sorts of decisions about which shops to visit, which ingredients to buy, and how long to cook the noodles, before realising that the chopstick requirement cannot be satisfied.

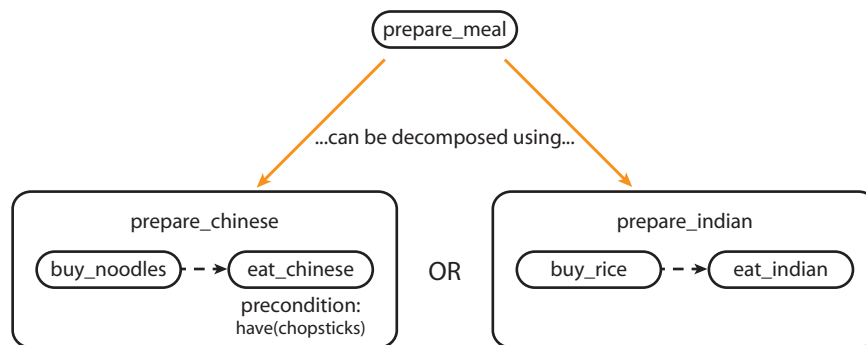


Figure 2.4: Example of HTN planning: preparing an exotic meal.

A number of planning systems use control strategies to guide search and stop the planner making “poor” choices of decomposition:

The *Task Formalism (TF)* of the early HTN planner *Nonlin* (Tate, 1977) allows the domain designer to specify several types of information about methods. For example, a *high level effect* of a method m specifies an effect that the body of m is designed to achieve. An *unsupervised condition* of m specifies a precondition of a descendant of m in the decomposition hierarchy that no other descendant is able to achieve. These pieces of information can help the planner prune inappropriate choices of method, but can affect soundness and completeness. For example, a high level effect of a method m may affect the soundness of a planner if one or more decompositions and linearisations of m do not have an appropriate concrete effect.

Similarly, *SHOP* and *SHOP-2* (Nau et al., 1999, 2003) allow human domain designers to annotate methods with the preconditions for situations in which they should be applied: a human designer could add a *requires_chopsticks* precondition to the the *prepare_chinese* method to prevent it being used when chopsticks are not available. This requires the planner to totally

¹³It is assumed that the agent cannot buy chopsticks when it is at the shops.

order tasks and decompose them in the order they will be executed, so that the complete world state is known at the beginning of each abstract task. SHOP requires method bodies to be totally ordered. SHOP-2 relaxes this requirement, creating a totally ordered plan from partially ordered methods.

Tsuneto et al. (1998) present a technique for automatically computing *external conditions* of methods from the decomposition hierarchy, by looking at the possible interactions between the descendant tasks of a method. External conditions serve the same purpose as Tate's unsupervised conditions, but because they are automatically calculated rather than designed by a human, they can be shown to never affect soundness or completeness.

Similar work by Clement and Durfee (1999a,b,c) is particularly relevant to this thesis as it forms the basis of the the research presented in Chapters 3 to 5. Clement and Durfee use *summary information* derived from methods and subtasks to detect possible conflicts directly between abstract tasks in non-recursive propositional HTN domains. They use this information both as a heuristic guide and to prune search states from which conflicts cannot be moved. Their planning mechanism, called *Concurrent Hierarchical Plans (CHiPs)*, is discussed in greater detail in Sections 2.2.3 and 3.2.

2.1.6 Local search planning

All of the planning algorithms described so far are refinement planning algorithms. They work by iteratively adding to a partial plan until a solution plan is found. At every iteration, the agent picks a refinement to apply to the current plan and uses various combinations of planning operators to create a set of new plans. The agent uses heuristics to *estimate* which new plan is most appropriate, and saves the others on an open list as *backtracking points*.

Backtracking guarantees completeness if the planner can always spot solution and failure states¹⁴. Additionally the search algorithm is able to signal failure if the entire search space has been covered and no solution has been found. The price of this is the storing of the backtracking points, which take up memory and can require maintenance in the event of external

¹⁴Failure states are states from which a solution cannot be reached. It is in these cases that the planner initiates backtracking.

changes to the environment.

Local search is an alternative technique in which few if any backtracking points are used. Search algorithms use a combination of the refinement operators from global search and *iterative repair* operators that correct individual flaws in the plan. Planners never backtrack: they simply keep changing aspects of their plans in an *ad hoc* manner until they find a solution.

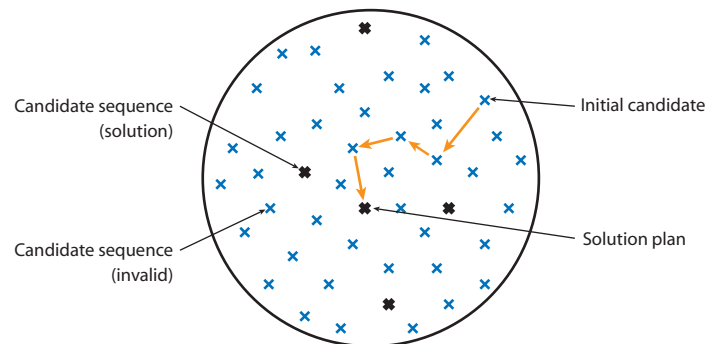


Figure 2.5: Local search in a space of candidate plans (adapted from Narayek, 2002).

Local search algorithms are “hill descending” algorithms¹⁵, constantly iterating towards “better” plans of smaller heuristic value, according to the local slope of a heuristic landscape. They do not guarantee completeness and normally only signal failure after being unable to find a solution for a predetermined length of time. Like all hill climbing algorithms, local search planners are prone to getting stuck in heuristic plateaux and local minima that do not contain solution plans. Local search can be effective, however, if good heuristics are used that create a relatively smooth landscape that minimises at solutions, and *stochastic search techniques* are used to help the planner escape local anomalies. Example techniques include:

Simulated annealing is a technique where the current location of search is loosely modelled as the position of a particle in a hot material that is slowly cooling. At first, when the material is at a high temperature, the particle will have lots of energy and will be able to move large distances. As the material cools, the particle’s energy will drop and it will eventually come to rest at the location of a local energy minimum. In search terms, this means that the agent starts with a candidate plan and randomly changes it to try and minimise the

¹⁵The term “hill descending” is used analogously to the common term “hill climbing”: hill descending algorithms search for conflict minima as hill climbing algorithms search for fitness maxima.

number of conflicts it contains. At first it will try large (“high temperature”) changes, but over time it will decrease the magnitude of its changes until a conflict minimum is reached. Hopefully this minimum will be low enough to constitute a solution to the problem.

Nogood constraints keep records of parts of plans that have failed to work together. They are similar to tabu lists, preventing agents from revisiting previous states, but only record information specific to conflicts and are kept around for the remainder of planning, rather than for a short period of time.

More information on stochastic search techniques can be found in Section 7.3.2 of Ghallab et al. (2004).

The *ASPEN (Automated Scheduling and Planning Environment)* framework for continual planning of spacecraft operations (Rabideau et al., 1999; Chien et al., 1999) uses local search to plan and replan in dynamic environments. The local search algorithms continuously “repair” the current plan in response to changes in goals and the environment. ASPEN is of particular interest because Clement has applied summary information techniques to it to create heuristics for scheduling abstract activities (Chapter 8 of Clement, 2002).

SAT planning (Chapter 7 of Ghallab et al., 2004) is a popular technique where planning is encoded as a *propositional satisfiability problem*, allowing the use of fast stochastic local search algorithms. SAT planning is the technique at the heart of the *BlackBox* planning system (Kautz and Selman, 1992), which was competitive with the fastest state space planners, including *Graphplan*, at the first AIPS Planning Competition in 1998. There are, however, arguments for avoiding SAT encoding and maintaining a planning based representation, both from an efficiency point of view and to facilitate development of new planning techniques (Brafman and Hoos, 1999).

Yokoo and Hirayama (2000) present several asynchronous algorithms for solving constraint satisfaction problems, which are discussed further in Sections 2.2.3 and 4.4.

2.2 Multi agent planning

This section describes various approaches to multi agent planning, building on refinement based and local search based approaches to single agent planning. The literature cited in this section is relevant to various subsets of the taxonomy of problems discussed in Section 1.1. Later chapters of this thesis concentrate more specifically on the subset of problems described in Section 1.2.

The multi agent systems described in this section follow the definition of “agent” introduced in Section 1.1.1. Multi solver and multi executive systems are not considered unless they also have a multi agent component. Similarly, multi agent systems are not considered that decompose problems “horizontally” into abstraction layers rather than “vertically” into subproblems (Section 1.1.1).

2.2.1 Centralised planning

Any of the techniques from Section 2.1 can be applied to multi agent planning if agents are willing to share complete information about their goals and plans. The process of centralised planning, shown in Figure 2.6, involves three steps:

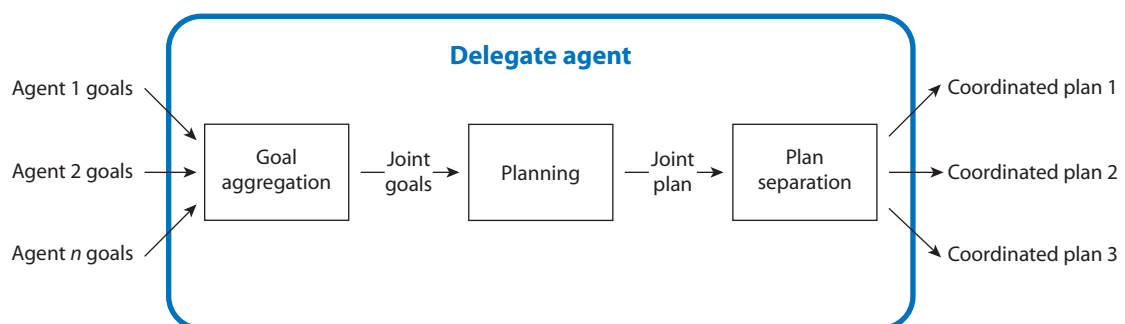


Figure 2.6: Flow of information in a simple centralised planning system.

Goal aggregation Agents’ goals are merged into a single set of *joint goals*, and passed to a *planning delegate* agent that takes on the job of planning on behalf of the team.

If goals are non-conflicting, aggregation can be as simple as unioning goals from each

agent¹⁶. If goals are conflicting or contradictory, agents may have to select a subset of goals that are achievable. Rosenschein and Zlotkin (1994) describe many bidding and negotiation protocols that can be applied to such problems. Their emphasis is on efficiency, simplicity, fairness and the prevention of deception and cheating (Zlotkin and Rosenschein, 1996). Domain dependent aspects of negotiation, such as agents' preferences for plans that achieve a maximum number of goals or that are robust to environmental events, are typically encoded as measures of “*cost*” or “*worth*” of goals and plans, which are fed into the protocols in a domain independent way.

Agents may not be able to determine whether or not goals are conflicting until some planning has been performed, meaning goal selection may have to be revisited after planning has started. Goal selection applies to single agent scenarios as well as multi agent ones: Beaudoin (1994) investigates this problem, which he calls “*meta-management*”, for a single agent in his PhD thesis. Without good algorithms to find compatible sets of goals this is a big problem: there are $\sum_{i=1}^n {}^n C_i$ selections from a set of n conflicting goals.

Planning The planning delegate creates a *joint plan* to achieve all the joint goals. Any of the single agent planning techniques in Section 2.1 can be used that have a sufficiently expressive plan representation.

Plan separation The joint plan must be split into a set of coordinated *individual plans* that can be passed to the relevant executives.

If “real world” plan execution is being considered, this step may involve reasoning about executives and their roles in the joint plan. Executives may have to be assigned to certain actions if the decision has not already been made during planning (Browning et al., 2004). Synchronisation and coordination information may also have to be inserted into the plan to ensure successful multi executive execution (Rosenschein, 1982; Biggers and Ioerger, 2001).

Centralised planning is an attractive approach because the planning delegate may draw on well established algorithms from single agent planning. However, centralised planning does have one major drawback in that individual agents have no privacy or independence. This may be

¹⁶Goal aggregation in HTN planning is essentially the merging of abstract plans (Section 2.2.2).

inappropriate if agents are acting on behalf of companies or have access to sensitive information (de Weerd and van der Krogt, 2002). In such cases it may be more appropriate for agents to keep their goals and knowledge private and only share information about their plans (Sections 2.2.2 and 2.2.3).

Centralised planning may also be inefficient if agents' goals do not conflict significantly. Korf (1987) categorises goals as *independent*, *serialisable* and *non-serialisable* depending on the nature of their relationships. Independent goals do not interfere during planning and are obviously solved faster by separate planners. However, serialisable and non-serialisable goals may interfere or conflict, making the best choice of approach less clear. This is discussed in more detail in the next section.

2.2.2 Plan merging

Plan merging is a popular approach to multi agent planning that addresses part of the problem of sensitive information and fares better than centralised planning when agents' goals are largely independent. Figure 2.7 shows a typical plan merging procedure:

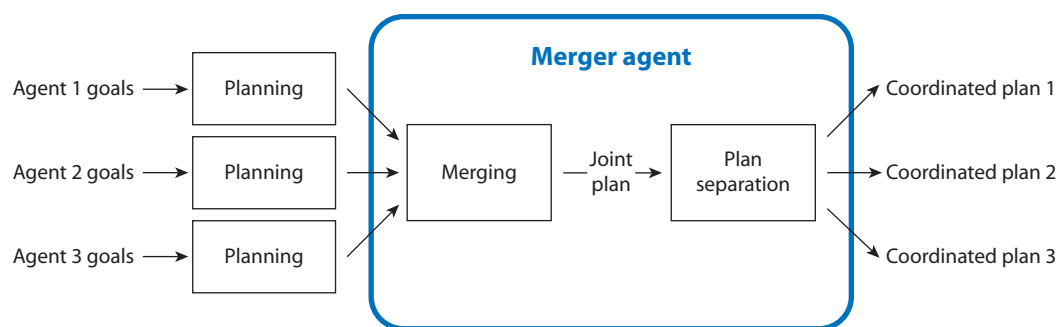


Figure 2.7: Flow of information in a simple plan merging system.

Planning is handled separately by each agent, in *isolation*, disregarding other agents and concentrating only on individual goals. The result is a set of uncoordinated individual plans that are not guaranteed to work when executed together, as actions can be rendered impossible and goals undone by other agents.

Plan merging All agents pass their uncoordinated plans to a *merger agent* for coordination

by *plan merging*. Plan information can safely be shared without exchanging information about the agent's original goals and knowledge, although agents must be willing to share the plans themselves before execution. The merger agent combines the uncoordinated individual plans to form a single coordinated joint plan, resolving conflicts by adding ordering constraints and variable bindings and by eliminating redundant actions (difficulties can arise if there are unresolvable conflicts in or between the individual plans: this is discussed in more detail below).

Plan separation is the same as in centralised planning, and is required to assign individual plans to executives.

Plan merging is also used in some planning systems when new goals are added after some planning has been completed. There are many examples of this in the field of *continual planning* where planning and execution are continual interleaved processes (desJardins et al., 2000).

Alami et al. (1998) present an “efficient forward chaining plan merging algorithm that supports multiple agents and continual planning”. At any point there are several agents with plans that have already been coordinated and are being executed. Agents periodically receive new goals from a central server and create uncoordinated plans to achieve them. Once an agent has a complete uncoordinated plan it requests and performs a *plan merging operation (PMO)*, in which it examines the coordinated plans from all the other agents and merges its new plan in with them. Agents can create new uncoordinated plans and execute existing coordinated plans concurrently and continuously, apart from during the short period of time in which a PMO is taking place. Only one agent is allowed to perform a PMO at a time, causing a potential bottleneck when goals are numerous and the plans to achieve them are short. Unresolvable conflicts between new and existing plans may occasionally make plan merging impossible, in which case a centralised planner is used as a fallback, pausing execution and recreating all plans from scratch.

Tsamardinos et al. (2000) present a plan merging algorithm for partial order plans that can handle simple conditional branching, quantitative time and actions with temporal duration. They use it to merge new goals into existing single agent plans. Plans are merged using constraint

satisfaction techniques on two constraint graphs: a *conditional simple temporal network*¹⁷, containing the start and end points of actions, and a *conflict-resolution graph* (Yang, 1997, cited by Tsamardinou et al.) of the threats on causal links in the plan.

de Weerd (2003) has developed a resource oriented representation of actions and state called the *Action Resource Formalism (ARF)*. He uses this as the basis of an anytime plan merging algorithm with polynomial complexity based on the exchange of resources between agents. The system requires agents to have an existing set of concurrently executable plans. Plan merging is not strictly required for the plans to be executable, but it may help reduce the cost of execution to the agents involved. Valk et al. (2005) present a related plan merging paradigm that adds an extra *pre-planning coordination phase*, in which agents exchange goals to suit their own planning knowledge. This helps create the set of independently executable plans required by de Weerd. The problem of allocating a set of planning tasks to a set of agents is shown to be NP-hard in general, but Valk et al. develop an approximate algorithm that suffices in most cases.

Disadvantages of plan merging Plan merging is a popular approach to plan coordination. However, it has two major drawbacks:

1. The success of the whole process is dependent on the ability of the merger agent to merge the individual plans into a coordinated joint plan. This may be impossible because of inter-plan conflicts that the merger cannot resolve, or because of a lack of time, information or memory capacity. If the merging phase fails, time spent planning will be wasted.
2. Individual agents cannot give each other assistance during planning because they plan in isolation. This means that, for example, an agent with exclusive control over a resource cannot assist other agents in their planning of related tasks. This can already be seen in the Airlock example in Figure 1.2.

The first disadvantage may partially be dealt with by providing the merger agent with an expressive planning formalism and flexible plan merging algorithms enabling it to resolve a wide

¹⁷A Conditional Simple Temporal Network is a variant of the well known Simple Temporal Network (Schwalb and Vila, 1998) that uses labels on nodes to represent the branches of a plan in which the relevant timepoints exist.

range of possible conflicts. The resolution of some conflicts may require alteration of existing single agent plans, which highlights potential issues of trust and misinterpretation. The second disadvantage, however, is more fundamental. To achieve this kind of cooperation agents must exchange information during planning and plan socially.

Serial solution of subgoals Korf (1987) did some influential work on the *serial solution* of subgoals in single agent planning, briefly mentioned in the last section, which has some relevance to plan merging and multi agent planning in general. Serial solution involves concentrating on one subgoal at a time: once a subgoal has been achieved, the relevant bits of state information are *fixed* so the planner cannot undo them, and the next subgoal is tackled. Serial solution is not possible for all sets of goals, but if it is possible it can significantly reduce problem complexity. Korf categorises subgoals in the following way:

Independent subgoals do not interfere with each other, and can be solved serially in any order.

For example, an agent could solve any number of “eight puzzles” independently: once each puzzle has been solved, it can be left untouched while the planner solves the others.

Serialisable subgoals can be solved serially in some orders but not in others. For example, an agent can put away its toys and then close the toy box, but cannot close the toy box and then put away its toys¹⁸.

Non-serialisable subgoals have to be solved at the same time. For example, the robots in the “airlock” problem in Figure 1.2 rely on each others’ movement to reach their destinations. It is impossible to move one robot and then the other.

Korf shows that the serial solution of n independent subgoals reduces planning time by a factor of n . However, with serialisable goals the results are not always clear cut, and with non-serialisable subgoals no speed increase is possible. In the majority of “interesting” planning domains, goals often have serialisable and non-serialisable relationships that make serial solution much more difficult. The relationships between goals may also be difficult to solve in general.

¹⁸If the agent were to close the toy box first, it would have to reopen it when it is putting away the toys. Undoing a previous goal is not permitted in serial solution.

Korf assumes that a single agent is doing all the planning, but his observations about goal types have some relevance for sets of concurrently planning agents. The problems being solved do not have to be strictly independent for plan merging to be subsequently possible, as some conflicts between plans (such as redundant actions) can be removed by the plan merging agent. However, individual problems do have to be independently solvable, so some distributions of goals between agents will be impossible to solve using plan merging.

Restrictions on subgoals Some attempts to solve this problem of independence have been made by limiting the interactions between the goals that may be given to different agents. For example, Yang et al. (1992) propose two domain independent restrictions that can be imposed on goals to ensure fast plan merging:

1. Given a set of plans S , the actions therein can be partitioned into a set of *mergeability classes* $\{E_1, E_2 \dots E_n\}$ such that actions can only be *merged* if they belong to the same class. The merging of a set of actions involves replacing them with a single action that has precisely the same effects.
2. In the joint plan formed by merging the members of S , if there is a precedence relationship such that an action a must be ordered before an action b from a different mergeability class, then the reverse precedence relationship of b before a cannot also be required.

These are limiting restrictions, but they still allow the representation of a significant number of planning problems (see the paper for examples). Yang et al. define two polynomial time algorithms for merging plans where each planning agent is given a single individual goal:

- One algorithm produces an optimal joint plan from a set of individual plans in $O(n^3)$ time.
- Another algorithm produces a near-optimal joint plan in situations where each planning agent produces a set of alternative individual plans. Again, the algorithm has $O(n^3)$ complexity.

Algorithms are also defined for situations in which planning agents produce plans for multiple goals. The algorithms ensure that plans can be merged if they can be found, but the assumption is still made that individual plans can be found without interaction during planning.

Implicit coordination and social laws Shoham and Tennenholtz (1995) show how *social laws* can be used to reduce conflicts in planning. Social laws are predefined rules that agents must follow. A simple example is the rule “*Always drive on the left.*”¹⁹. If all driving agents follow this rule they rarely need to worry about collisions. Social laws are an effective means of reducing time spent on planning, but poorly defined rules can prevent planning being sound. For example, if the left hand lane is blocked then it is a reasonable course of action to temporarily cross to the other side of the road: the rule defined above forbids this, so agents would be unable to create a valid plan in this case.

Briggs and Cook (1996) extend the social law paradigm by allowing agents to relax social laws when necessary to find a solution. Laws are given a ranking, from the most strict to the most flexible. If agents cannot find valid plans with the strictest set of laws, they relax the highest ranking laws and try again. Soundness is preserved because agents end up trying to find plans subject to *no* laws, which equates to “normal” planning and plan merging. Optimality of plans is not guaranteed if they are created following laws, but in complex domains the quick production of a sub-optimal plan is often preferred to the time consuming production of an optimal one. Briggs and Cook also devise a system whereby agents can acquire suitable social laws using machine learning, although this is beyond the scope of this thesis.

2.2.3 Distributed planning

In complex “real world” environments, agents with different goals and abilities may need to create plans quickly and individually without relying on other agents for help²⁰. In some situations, demands for independence or privacy may limit the applicability of plan merging techniques. In these situations agents may require a higher degree of independence than that provided by plan merging.

An alternative to plan merging is to completely distribute planning. In this approach, agents are allowed to communicate and exchange information during planning as shown in Figure 2.8. While this approach is flexible, it is also complicated as each agent has to keep other agents informed of salient changes to its plan while staying on top of similar reciprocal messages.

¹⁹Driving on the right is also a valid option.

²⁰It is useful to be able to seek assistance from other agents but restrictive to require it.

There are two types of distributed search only one of which provides independence but both of which are mentioned in the literature²¹:

Distributed global search is essentially a form of centralised planning that takes advantage of concurrency. A single agent controls the search process and assigns refinements to other agents as planning tasks. The results of refinements are passed back to the delegate and stored ready for the next iteration. This is the paradigm adopted by, for example, the *Multiagent Planning Architecture (MPA)* of Wilkins and Myers (1998).

Agents in this paradigm are equivalent to subordinate *problem solvers* (Section 1.1.1) because they are directly under the control of a master agent. This is potentially a way of increasing the speed of centralised planning, but it does not deal with the issue of independence any more than centralised planning does.

Distributed local search is an approach in which agents plan independently but periodically update each other with salient information about resource usage, requests for planning assistance and so on (Figure 2.8). This approach focuses on the independence of the agents, but means that centralised control like that in centralised planning and plan merging is impossible. Refinement planning algorithms are of limited use to agents in this paradigm because changes in other agents' plans, which are essentially changes in the external environment, can invalidate established parts of the plan. Agents are forced to use alternative algorithms, such as local search algorithms (Section 2.1.6), that do not rely on a static external environment.

Distributed local search is a topic that has been dealt with very little in the planning literature, although relevant approaches have been used in the field of *distributed constraint satisfaction*. Yokoo and Hirayama (2000) present several asynchronous algorithms for solving constraint satisfaction problems with single and multiple agents. These algorithms do not store backtracking states, relying instead on *nogood constraints* to store invalid combinations of values so that they are not visited again. These algorithms and their application to planning are discussed in more detail in Section 4.4. Brenner (2003) has also suggested basing a forward chaining algorithm

²¹The names quoted here are not in common use: they are merely for reference within this thesis.

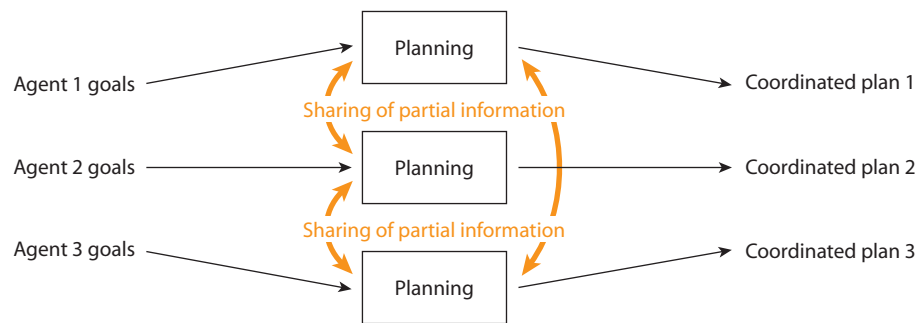


Figure 2.8: Flow of information in a simple distributed local planning system.

for multi agent planning on asynchronous backtracking, although at the time of writing his algorithm is unpublished.

Commitments and conventions Jennings (1993) creates a general theory of multi agent problem solving using a distributed goal search formalism. He argues that multi agent cooperation and coordination are based on two central premises:

Commitments are “*pledges to undertake a specified course of action*”. Commitments about future actions are fundamental because without them consensus cannot be achieved. For example, if an agent commits that it is going to a specific bar in the evening, other agents can decide whether to join in and when and where to meet up: if no commitment is made, no plan will be made either²². Commitments need to be, within reason, binding. If they are rarely held for long, agents cannot reliably use them as the basis of further decisions. Refinements in planning are essentially commitments to particular plan features. For example, an agent may commit to a certain order of execution or to the use of a particular resource. When agents backtrack or try different branches of search they retract or switch between commitments, which can be bad for other agents. This is discussed further in Section 4.4.

Conventions “*provide a means of monitoring commitments in changing circumstances*”. Changing circumstances sometimes require commitments to be broken. Agents need to agree on when this is necessary, and on the courses of action to choose in such an event.

²²It is expected that many readers will be familiar with this situation.

Random changes to commitments are equivalent to unpredictable events in a dynamic environment (Section 1.1.4): they are more of a hindrance than a help.

In planning, conventions can be enforced in a number of ways, including distributed search algorithms, negotiation protocols and social laws. The important feature of all of these techniques is *predictability*: this is what distinguishes a change in an agent's plan in a well thought out multi agent planning approach from a random change in the external environment.

Jennings performs an in-depth analysis of commitments and conventions in various hypothetical scenarios, and shows how they can be used to model existing distributed problem solving systems. His analysis is based on a *distributed goal search* formalism. An example is shown in Figure 2.9. Agents' plans are modelled as classic and/or goal trees (Chapter 7 of Russell and Norvig, 1995). It is possible for agents to share goals and goals to share subgoals. Further interdependencies can exist between goals belonging to the same agent or different agents. These interdependencies can be *strong* ("the results of $goal_a$ are required to achieve $goal_b$ ") or *weak* ("the results of $goal_a$ might help to achieve $goal_b$ "), unidirectional or bidirectional. Interdependencies can also exist between goals and resources: resources are either required to achieve a goal or are provided by achieving a goal.

Given a distributed goal tree and a set of interdependencies, agents make commitments by choosing *or* branches to pursue and imposing constraints on the order in which they will be pursued. The structure of the interdependencies helps agents determine which commitments are likely to be the most appropriate and/or stable. Jennings's goal trees are rich and expressive formalism that has inspired a number of projects in distributed artificial intelligence:

Partial Global Planning *Generalised Partial Global Planning (GPGP)* (Decker and Lesser, 1992) is concerned with the "*distributed coordination problem*", which is described as "[*how*] the local scheduling of activities at each agent [should be] affected by non-local concerns and constraints". GPGP uses a goal formalism called TÆMS (Horling et al., 1999) that is very similar to Jennings's goal trees: quantitative interdependencies can be specified between goals and resources at varying levels of abstraction, allowing agents to schedule problem solving activities to maximise the quality of their results. GPGP is a family of coordination algorithms

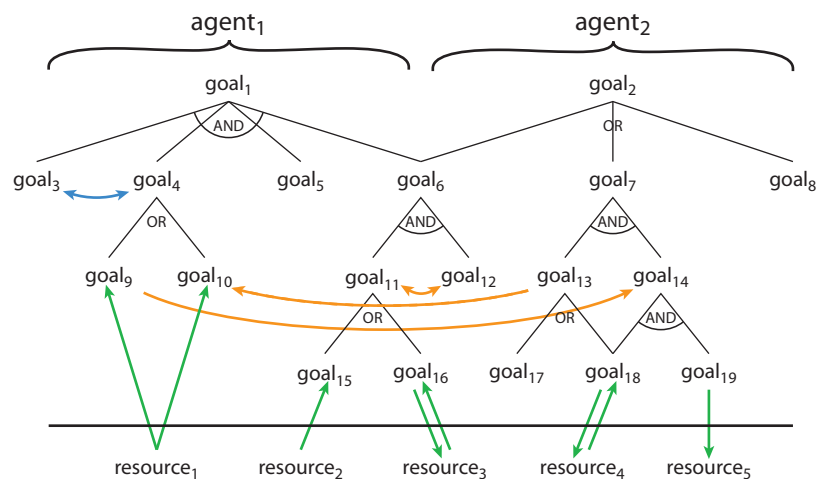


Figure 2.9: A distributed goal search tree adapted from (Jennings, 1993) involving two agents: $agent_1$ and $agent_2$. Black lines represent and/or decomposition of goals, blue arrows represent interdependencies between goals of a single agent, orange arrows represent interdependencies between goals of different agents, and green arrows indicate interdependencies between resources and goals.

for goal allocation and subdivision, information sharing, and result merging.

GPGP does not deal explicitly with planning problems in the conventional sense: agents are provided with a problem structure in terms of a goal tree, whereas in planning the tree would have to be inferred from the description of the problem domain. The quantitative nature of GPGP gears it towards optimisation problems, in which there are few hard constraints but many weak constraints that affect the quality of the solutions output.

Task trees and summary information The work of Clement and Durfee (1999a,b,c), which was briefly mentioned in Section 2.1.5, is also based on a goal tree like formalism. Their *task trees* are a blend of goal trees and HTN planning formalisms, and their *summary information* allows weak and strong interdependency information to be automatically generated from the planning domain. In performing the crossover with HTN planning Clement and Durfee drop some of the flexibility of Jennings's representation. For example, there is no multiple parent-hood as there is with $goal_6$ and $goal_{18}$ in Figure 2.9, and this means that a single task cannot be controlled by more than one agent. This simplification is presumably in the interests of simplicity when developing the formal semantics of task trees, although this is not made clear in the literature.

Task trees and summary information form the basis of the *MPF* planning mechanism used in this thesis, and are described in detail in Section 3.2.

Jennings's hypothesis about the centrality of commitments and conventions makes many state space planning approaches less attractive from the point of view of distributed local planning:

- Agents need to be able to make commitments progressively during planning. If agents do not make commitments regularly or retract them too quickly, other agents cannot make decisions based on them. Agents using planning graphs, in particular, do not commit to any temporal orderings until the very end of planning, when a valid plan is extracted from the graph (Section 2.1.3). This gives a single agent enormous amounts of flexibility, but is bad from a multi agent point of view.
- Distributed local planning is about resolving conflicts between agents' plans. An agent may be required to resolve a specific conflict at a specific stage in planning, which is more likely to be possible if agents only commit to plan features when necessary to remove conflicts that have already been detected. State space planning algorithms commit heavily to temporal orderings regardless of whether there are conflicts in the plan or not, meaning they may have to backtrack several levels if there is a problem caused by an action in the middle of a plan. Plan space approaches that use the principle of least commitment are more likely to be able to make a quick change to fix a particular conflict.

2.3 Summary of multi agent planning approaches

This chapter has introduced many approaches to single and multi agent planning. In particular, three general types of multi agent planning approaches have been outlined:

Centralised planning collects individual planning problems into one large joint planning problem (as does distributed global planning), and passes that problem to a single agent to solve. Independence of individual agents is sacrificed in exchange for the ability to use fast refinement planning techniques.

Plan merging techniques allow agents to plan independently using single agent planning techniques. The individual plans are collected by a single agent and merged into a coordinated joint plan. Individual agents are more independent in this approach than in centralised planning, but the division of search into two separate phases causes it to be inappropriate for some problems. Refinement planning approaches are useful throughout, although the plan merging agent may benefit from increased flexibility if it is given the ability to undo commitments made in individual plans to create a valid joint plan (Section 6.1.4): such ability may be better implemented using local search techniques.

Distributed local planning approaches give agents the maximum independence possible: they only share the information necessary to coordinate their plans, and do not rely on the outsourcing of planning or merging activities to others. The distributed nature of search means that refinement search algorithms are inappropriate without modification. Local search provides a possible alternative paradigm.

In terms of independence, distributed planning is clearly the most attractive of the three approaches. However, changes in other agents' plans effectively cause a chain of external events that violate the central assumption of a static world that is associated with refinement planning approaches. Special search techniques are required to make distributed local planning viable.

The remainder of this thesis concentrates on the implementation and empirical comparison of versions of these three approaches. This work clarifies and reinforces the issues and arguments discussed this far, and provides a deeper understanding of the problems at hand. It is hoped that this exercise will help to examine the advantages and limitations of each approach, and provide pointers for the development of good approaches for multi agent planning.

A common planning mechanism is required on which the three approaches above can be implemented: this will remove extraneous differences between the approaches and make an empirical comparison between them as fair as possible. Chapter 3 develops the *Multi agent Planning Formalism (MPF)*, the planning mechanism used throughout the rest of this thesis. MPF is based on the *Concurrent Hierarchical Plans (CHiPs)* of Clement and Durfee (1999a,b,c), which is itself based on a fusion of Jennings's distributed goal trees and HTN planning.

Chapter 4 develops algorithms, based on MPF, for centralised planning, plan merging, and distributed local planning. The centralised planning and plan merging algorithms use a straight-forward refinement based search. The distributed local planning algorithm is a blend of local refinement planning and distributed constraint satisfaction techniques for the removal of inter agent conflicts. Chapter 5 investigates the performance of the three approaches when applied to a number of “traditional” planning problems. Chapter 6 concludes the thesis by revisiting the contributions from Section 1.3 in the light of the rest of the thesis and outlining possible directions for future research.

Chapter 3

A common planning mechanism

The performance of any computer system is partially dependent on the manner of its implementation. For a given problem, the performance of a multi agent planning approach will depend on many factors including: the details of the approach itself, the single agent planning algorithm it uses (if any), the programming language of implementation, the method used for evaluation, and the representations, refinements and operators on which it is based. The details of the planning and coordination algorithms used will necessarily be different in different approaches, but to perform an accurate comparison all other details should be as fixed as possible. This chapter presents a common *planning mechanism* (Section 2.1.1), on which implementations of the three approaches in Section 2.2 are built in Chapter 4.

The planning mechanism used in this thesis, called the *Multi agent Planning Framework (MPF)*, is based on the *Concurrent Hierarchical Plans (CHiPs)* of Clement (2002). CHiPs is an HTN planning mechanism developed for the coordination of plans for multiple executives¹. It uses *summary information* to detect conflicts directly between abstract tasks instead of having to decompose to the level of primitive tasks. This allows the early detection of solution and failure states, pruning branches of search early on and finding solutions more quickly than HTN planners that rely on primitive preconditions and effects (Section 2.1.5).

Two varieties of MPF are described below: *propositional MPF (pMPF)* and *first order MPF (fMPF)*. pMPF is very similar to CHiPs, while fMPF introduces a number of novel extensions,

¹Clement refers to this differently, as “multi agent planning”, as a result of the terminological ambiguities described in Section 1.1.1.

making it useful for the representation of larger, more complex problems. fMPF is used in most of the experiments in Chapter 5.

Section 3.1 discusses some of the key requirements for a multi agent planning mechanism, and Section 3.2 provides an overview of the key features and limitations of CHiPs. Sections 3.3 and 3.4 discuss pMPF in detail, and Sections 3.5 and 3.6 discuss the extensions present in fMPF.

A semi-formal notation is used to describe data structures and algorithms throughout this chapter. A guide to the notation can be found in Appendix A.

3.1 Requirements

A number of features are desirable in a planning mechanism to aid both the representation of multi agent problems and the use of the mechanism in multi agent planning approaches. This section outlines the criteria that lead to the adoption of CHiPs as a main source of inspiration and the development of MPF as a planning mechanism. Other requirements for multi agent planning mechanisms have since been identified as a result of the development of algorithms in Chapter 4 and the empirical work in Chapter 5. These additional requirements are discussed in the conclusions in Section 6.1.3.

3.1.1 Joint and multi executive plans

As mentioned in Section 1.1.1, the terms *individual plan* and *joint plan* are used ambiguously in the literature, either referring to plans made by single and multiple agents or plans for single and multiple executives. As this thesis is concerned with planning rather than plan execution, the following definitions are used:

- an *individual plan* involves tasks for achieving the goals of a single agent;
- a *joint plan* is the (explicit or notional) combination of the individual plans of several agents.

Several individual plans can be merged into a single joint plan, and a joint plan can be split up into a set of coordinated individual plans. Similarly, it is possible for both individual and

joint plans to contain information about multiple executives: a multi executive plan can be split up into a set of coordinated single executive plans, and several single executive plans can be merged into a single joint executive plan.

Any multi agent planning system needs at least to be able to differentiate and convert between individual plans and joint plans. If plan execution is significantly complex, conversion between single and multi executive plans may also be important.

The empirical work in this thesis ignores plan execution to concentrate on planning (Section 1.2). It is sufficient and convenient in this case to model executives as part of the planning problem (or to ignore them altogether). For example, in a multi gripper Blocksworld problem the action to move a block may be represented as:

$$\text{move}(\text{?block}, \text{?src}, \text{?des}, \text{?gripper})$$

where *?gripper* is a variable that can be bound to a particular gripper executive during planning. With this simple model, a multi executive plan is simply a plan containing concurrently executing tasks.

Agents are more difficult to represent than executives as they have complex internal state, part of which consists of plans themselves. Fortunately, explicit modelling of other agents' mental state is only necessary in certain types of advanced agent architecture. When only planning is being considered, a simpler mechanism will suffice: plans must be annotated with a minimum amount of information to represent *ownership*, identifying the agent or agents responsible for individual constraints, goals or actions.

3.1.2 Expressive temporal model

While multi agent planning does not necessarily require temporal reasoning of the complexity found in *Zeno* (Section 2.1.4; Penberthy and Weld, 1994), it does require a model capable of representing concurrent actions, preconditions and effects. Agents need to be able to commit to temporal orderings in response to aspects of other agents' plans as well as their own. This makes temporal flexibility very important.

Some recent single agent planners have used temporal models based on *temporal constraint networks* of various kinds (Dechter et al., 1991). For example, *Simple Temporal Networks (STNs)* are used by many planners to provide a simple quantitative temporal model. MPF is based on *interval temporal algebra* (Allen, 1983). While interval algebra does not provide qualitative information, it does allow qualitative reasoning about discontinuous disjunctions of temporal orderings with relatively little computation. This provides a powerful plan representation in which diverse sets of temporal orderings can be simultaneously represented (Section 3.3.4).

3.1.3 Flexible refinement

As discussed in Section 2.1.1, agents typically plan by iteratively making commitments in the form of *refinements*. An agent chooses a feature to refine, enumerates planning operators that may be used to do the refinement, and then chooses the operator that produces the “best” plan for the next iteration of the algorithm according to some heuristic.

The *flexibility* of a strategy for refinement choice becomes important in situations where agents impose changing constraints on each other. Consider two agents, *Alice* and *Bob*, deciding how to get to work:

1. Bob has a big meeting in the morning requiring lots of heavy papers. He decides he will take the car to work.
2. Alice has to go to the shops. If Bob has the car she cannot go, because she needs the transport to carry the shopping home.
3. Bob changes his mind and postpones his meeting until the next day. He reconsiders his use of the car and decides to walk to work.
4. Alice is left getting up early and walking to work for no reason. She has to revise previous decisions if she wants to get a better plan.

Plan revision is unavoidable in some cases. For example, Bob’s decision not to use the car may be based on new information which he did not have when he started planning: in fact, planning itself may reveal derived constraints that require agents to rethink previous decisions.

However, agents should be able to identify the constraining parts of their plans so they can *commit* to shared resource usage, avoiding unnecessary backtracking (Jennings, 1993). *Resources* comprise any part of the planning problem that can be changed or manipulated by a plan: *shared resources* are resources that can be altered by multiple agents. In this case, the car is the constraining shared resource and the agents should be able to choose refinements accordingly, identifying a suitable policy on shared resources early on and preventing unnecessary backtracking.

Refinement selection should be driven by resource usage rather than some other factor. While new parts of a plan will be chosen so that they are compatible with existing parts, later backtracking may cause conflicts and the planning algorithm will need to be able to reason directly about these flaws.

3.1.4 Accurate heuristics

As discussed in Section 2.1.6, when external constraints change unpredictably, global search becomes increasingly difficult and agents have to rely on local search algorithms that do not store information in the long term. Even in static environments, concurrently planning agents may need local search mechanisms to cope with unexpected changes in each others' plans. Because local search algorithms can easily get stuck in local minima and plateaux (Section 2.1.6), accurate heuristics are needed that produce smooth heuristic landscapes with few local minima.

3.2 Overview of task trees and summary information

This Section provides a brief overview of the approach taken in CHiPs and why it satisfies many of the requirements from the previous section. It also outlines some limitations of CHiPs and novel approaches for overcoming them. Many technical details are left until later sections.

CHiPs is an HTN mechanism which allows the detection and resolution of conflicts not only at the level of primitive tasks, but also at arbitrary levels of abstraction. This is done by computing and reasoning about *summary information* about the resource usage of abstract tasks. Summary

information can be used to predict conflicts that may occur later in planning, smoothing the heuristic landscape and improving the accuracy of search.

The following example will be used in the next few sections to show how summary information is calculated and used:

Consider a simple problem based on the dilemma of Alice and Bob (Section 3.1.3).

Bob needs to get to work and has the choice of walking or taking the shared car.

Alice needs to get to the shops later on and needs the car to get there.

Disregarding for the moment that Alice and Bob may be separate agents, and modelling them simply as executives within a centralised planning system, a single joint plan can be constructed involving tasks for both executives as shown in Figure 3.1. For the purposes of the example it is assumed that all tasks in these method bodies are primitive². Their preconditions and effects are shown in Figure 3.2.

$$\begin{aligned}
 plan &= \langle \text{tasks: } (bob_to_work, alice_to_shops) \rangle \\
 method_1 &= \langle \text{head: } bob_to_work, \text{body: } \langle \text{tasks: } (bob_get_in_car, bob_drive_work, \\
 &\hspace{15em} bob_get_out_car) \rangle \rangle \\
 method_2 &= \langle \text{head: } bob_to_work, \text{body: } \langle \text{tasks: } (bob_walk_work) \rangle \rangle \\
 method_3 &= \langle \text{head: } alice_to_shops, \text{body: } \langle \text{tasks: } (alice_get_in_car, alice_drive_shops, \\
 &\hspace{15em} alice_get_out_car) \rangle \rangle
 \end{aligned}$$

Figure 3.1: Simple example of an HTN planning problem involving two executives. All actions and state literals are propositional and all tasks are totally ordered as listed, making temporal and binding constraints redundant.

3.2.1 Task trees

A traditional HTN planner such as UMCP (Erol, 1996) would attempt to solve this problem by iteratively choosing an abstract task from *plan* and substituting for it using the body of an appropriate method. The obvious solution is to decompose *bob_to_work* with *method₂*

²Readers who are unfamiliar with HTN terminology may find it useful to refer back to Section 2.1.5.

$$\begin{aligned}
& \text{preconditions}(\text{bob_get_in_car}) = \{\text{car_at_home}\} \\
& \text{effects}(\text{bob_get_in_car}) = \{\text{bob_in_car}\} \\
& \text{preconditions}(\text{bob_drive_work}) = \{\text{bob_at_home}, \text{bob_in_car}\} \\
& \text{effects}(\text{bob_drive_work}) = \{\neg\text{bob_at_home}, \text{bob_at_work}, \\
& \quad \neg\text{car_at_home}, \text{car_at_work}\} \\
& \text{preconditions}(\text{bob_get_out_car}) = \{\text{bob_in_car}\} \\
& \text{effects}(\text{bob_get_out_car}) = \{\neg\text{bob_in_car}\} \\
& \text{preconditions}(\text{bob_walk_work}) = \{\text{bob_at_home}\} \\
& \text{effects}(\text{bob_walk_work}) = \{\neg\text{bob_at_home}, \text{bob_at_work}\} \\
& \text{preconditions}(\text{alice_get_in_car}) = \{\text{car_at_home}\} \\
& \text{effects}(\text{alice_get_in_car}) = \{\text{alice_in_car}\} \\
& \text{preconditions}(\text{alice_drive_shops}) = \{\text{alice_at_home}, \text{alice_in_car}\} \\
& \text{effects}(\text{alice_drive_shops}) = \{\neg\text{alice_at_home}, \text{alice_at_shops}, \\
& \quad \neg\text{car_at_home}, \text{car_at_shops}\} \\
& \text{preconditions}(\text{alice_get_out_car}) = \{\text{alice_in_car}\} \\
& \text{effects}(\text{alice_get_out_car}) = \{\neg\text{alice_in_car}\}
\end{aligned}$$

Figure 3.2: Preconditions and effects of primitive tasks from Figure 3.1.

and *alice_to_shops* with *method*₃, as this resolves issues over the availability of the car. If *bob_to_work* is decomposed with *method*₁, a conflict would inevitably result as there would be no way to achieve the precondition *car_at_home* of *alice_drive_shops*. This is an example of the conflict detection problem discussed in Section 2.1.5: the planning agent may initially not be able to decide which method to choose. While the amount of backtracking required here if *method*₁ were picked first would be small, it is possible to imagine situations where the cost of backtracking may be very high.

CHiPs approaches the problem in a way that, in this case, avoids backtracking. First of all, a *task tree* is built that explicitly represents the possible decompositions of abstract tasks in the plan. Task trees are *and/or trees* similar to the *goal trees* suggested by Jennings (Section 2.2.3). A tree for the example problem is shown in Figure 3.3.

The root of the task tree (level 0) is a task network representing the plan. Level 1 nodes are tasks in the plan, level 2 nodes are task networks representing decompositions of level 1 nodes, level 3 nodes are tasks in level 2 networks and so on. Decomposition becomes the process of

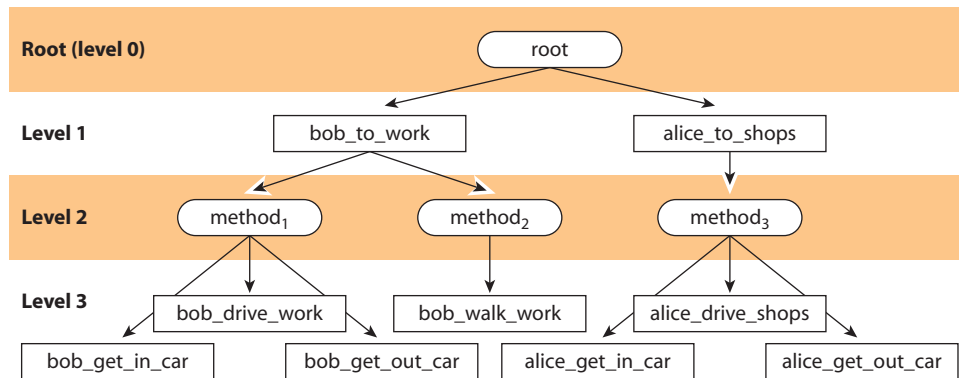


Figure 3.3: Simplified task tree for the abstract task and methods in Figure 3.1. Task networks are shown as capsules with rounded corners. Tasks are shown as rectangles. Arrows indicate membership (task network \rightarrow task) and possible decomposition (task \rightarrow task network).

selecting a level 1 task t that has one or more children(t), selecting one of its children d , and replacing t with children(d) while making appropriate updates to constraints on timings and resource usage³.

3.2.2 Histories

A task tree is essentially a partial plan representing a disjunction of possible decompositions, temporal orderings and variable bindings. A *history* is a complete decomposition, linearisation and grounding of a task tree; each tree represents a set of possible histories, each of which may or may not be a solution to the planning problem⁴. Every node in a task tree is associated with its own set of histories that can be formed from its subtree.

Abstract solutions and failures A task tree is referred to as an *abstract solution* if every history derivable from it is a conflict free solution to the planning problem. Similarly, a task tree is referred to as an *abstract failure* if none of its histories are conflict free solutions. Because subtasks can be interleaved after decomposition, the number of possible histories grows exponentially as the height of the tree increases. This makes the detection of abstract solutions and failures difficult for non trivial task trees. CHiPs is nevertheless capable of detecting some abstract solutions and failures through the use of summary information.

³A precise definition of the children function for MPF task trees is given in Section 3.3.6.

⁴Histories are a mechanism-specific interpretation of the *candidate plans* introduced in Section 2.1.1.

3.2.3 Summary information

The leaves of a task tree represent primitive tasks with preconditions and effects specified in the initial problem description. The possible preconditions and effects of nodes higher up the task tree can be calculated by propagating summary information from the leaves upward. Summary information is specified using three types of structure, referred to collectively as *summary conditions*:

Summary preconditions are net preconditions that are present in one or more histories of a node.

Summary postconditions are net effects that are present in one or more histories of a node.

Summary inconditions are states that are visited during one or more histories of a node, but are not summary preconditions or postconditions (Section 3.3.7). For example, if Bob travels to work in the car, but starts and finishes the journey on foot, *bob_in_car* is an incondition of the journey as a whole.

Existence of summary conditions Summary conditions are classified as *must* or *may* conditions, providing an estimate of whether they occur in all or a subset of the histories of a node. *Must* conditions appear in all histories of a node, making them definite features of any plan produced; *may* conditions appear in some histories but not others, meaning that they may or may not appear in a final plan.

Figure 3.4 shows the task tree from Figure 3.3, annotated with example summary information. Note the presence of inconditions in the summary information for *method₁* and *method₃*, and the presence of *may* conditions in the summary information for *bob_to_work*. Note also that some summary conditions are merged as they are propagated up the tree, so some information is lost during summarisation: summary information is a *relaxed* representation of the possible resource usage of a task or task network.

Relationships between summary conditions Relationships such as *achieving*, *clobbering* and *undoing* that can be found between preconditions and effects at the primitive level, can

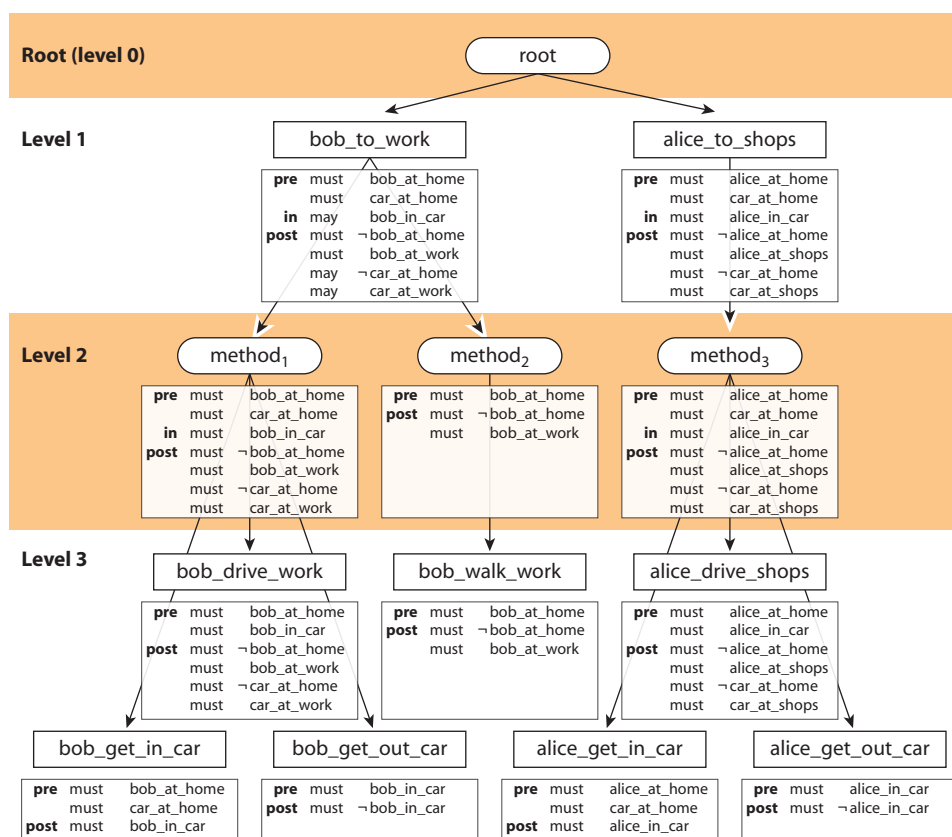


Figure 3.4: Summary information for the task tree in Figure 3.3. The tasks *bob_get_in_car*, *bob_drive_work* and *bob_get_out_car* occur in order in the plan, as do the tasks *alice_get_in_car*, *alice_drive_shops* and *alice_get_out_car*.

also be found between summary conditions at arbitrary levels of abstraction. This information can be used for the detection of some (but not all) abstract solutions and failures. Conflicts between summary conditions of level 1 nodes are called *threats*, some of which can be resolved by decomposition and temporal ordering and some of which cannot. A task tree with no threats is an abstract solution to the problem; a task tree with one or more unresolvable threats is an abstract failure.

In problems with propositional actions and world state, most of the summary information can be calculated once prior to planning and cached for later reuse.

Complexity of summarisation Clement (2002) states that the complexity of calculating the summary information of a node from the summary information of its children is $O(d^2s^2)$, where d is the number of child nodes and s is the number of summary conditions per child. From this

he shows that, in level i of a task tree of height h , with an average d children and c (non-summary) preconditions and effects per node, the complexity of summary information is as follows (Chapter 6 of Clement, 2002):

1. d^i nodes in the level
2. $O(d^{(h-i)}c)$ summary conditions per node (assuming the worst case where summary conditions do not merge during propagation)
3. $O(d^{(2h-i)}c^2)$ operations to derive the summary information for level i from the information for level $i + 1$
4. $O(d^{2h}c^2)$ operations to detect conflicts between summary conditions in level i
5. $O(k^{d^i})$ possible orderings of the tasks in level i (where k is a constant)
6. $O(k^{d^h})$ possible orderings of the leaf nodes in the tree (level h)

The two important results here are that the complexity of checking for conflicts between summary information is independent of level in the tree (item 4), and as long as the tree does not have to be expanded to the primitive level, the complexity of finding a valid task ordering is reduced by a factor of $O(k^{(d^h-d^i)})$ (items 5 and 6).

3.2.4 Heuristics and test functions

Rather than relying on primitive preconditions and effects to detect conflicts and assess plan quality, agents using summary information can reason directly about threats in the task tree. For example, an agent examining summary information for the task tree in Figure 3.3 would be able to predict the conflict that might occur over cars and choose decompositions accordingly.

Several useful heuristics and test functions can be automatically derived from summary information:

Abstract detection of solutions (Clement's `can_any_way` function; Section 3.4.1) If there are no threats on summary preconditions of level 1 children, then any history derived

from the task tree will be conflict free and will thus be an abstract solution. This function is used as the solution test function described in Section 2.1.1.

Abstract detection of failures (Clement’s *might_some_way* function; Section 3.4.1) If an unresolvable threat exists between two *must* summary conditions, then every history derived from the task tree will contain an equivalent conflict. The tree will therefore be an abstract failure. Abstract failures are pruned from search, preventing the agent wasting time search through possible derivatives.

Quantitative comparison of possible refinements The “importance” of a refinement can be measured in terms of the number of threats it has the potential to resolve. This allows the adoption of an early resolution strategy for refinement choice based upon inter-agent and intra-agent conflicts, as described in Section 3.1.3.

Quantitative comparison of candidate plans The “quality” of a plan can be measured in terms of the number of resolvable threats it contains. Agents can choose the order in which to visit plans output by refinements to minimise the number of threats encountered.

Joint and individual heuristics The information above may be generated jointly for the whole team or individually for each agent. *Joint* functions count every threat between every pair of summary conditions in the joint plan, whereas *individual* functions only count threats where the clobbered or unachieved summary condition belongs to a specific agent. The choice of joint or individual functions is important in approaches such as distributed local planning (Section 4.4) where agents have access to information about their own plans and the plans of others.

3.2.5 Advantages and limitations of the approach

CHiPs fulfils many of the requirements specified in Section 3.1, each of which is revisited below:

Joint plans can be represented within a single task tree. With small modifications, individual

plans can be represented in separate task trees. Individual task trees can be coordinated through the sharing and analysis of level 1 summary conditions, together with minimal information on timings.

Temporal constraints are attached to task network nodes in the tree. Various models, including interval temporal algebra, STNs and DTNs, can be represented with little effect on the rest of the planning system.

Refinements are chosen on the basis of threats between summary conditions. Inter- and intra-plan threats can be resolved early or late depending on the strategy for refinement choice used by the planning agent.

Detailed heuristics are provided at all levels of abstraction, allowing agents to make informed decisions about which plans and refinements to choose to minimise backtracking during distributed search.

MPF adds novel features to CHiPs to address four limitations:

Task ownership While CHiPs is capable of representing single and multi agent plans in a single task tree, it does not contain suitable information to support the coordination of several individual task trees. Extra annotations are required to represent task and constraint ownership.

Flexible temporal representation The temporal reasoning used in MPF is a combination of *interval temporal algebra* (Section 3.3.4; Allen, 1983) and novel reasoning about “*temporal envelopes*” (Section 3.3.8), as opposed to the *point temporal algebra* of CHiPs. This allows greater temporal flexibility in the plan representation, allowing decisions about temporal orderings to be deferred until later in planning and reducing the branching factor of the search algorithm.

First order state representation Actions and state literals in CHiPs are propositional. First order problems can, of course, be compiled into propositional format, but at the cost of an exponential increase in the size of the task tree. This is discussed in detail in Section 3.5.

Recursive problems The introduction of first order actions and literals to CHiPs is not without its problems. In some domains with first order actions, some tasks can be decomposed into abstract histories involving other actions of the same type. Consider, for example, the Blocksworld problem in Figure 3.5. In the figure, the $clear(?x)$ subtree can be of arbitrary size depending on the number of blocks.

Tasks are *recursive* if they have an *action* and can be decomposed into abstract plans containing instances of *action*. It is difficult to predict a bound on tree size in general when recursive tasks are present, and accurate summary information cannot be generated without generating a complete task tree prior to planning (although this is the subject of ongoing research; Gurnell, 2004)⁵.

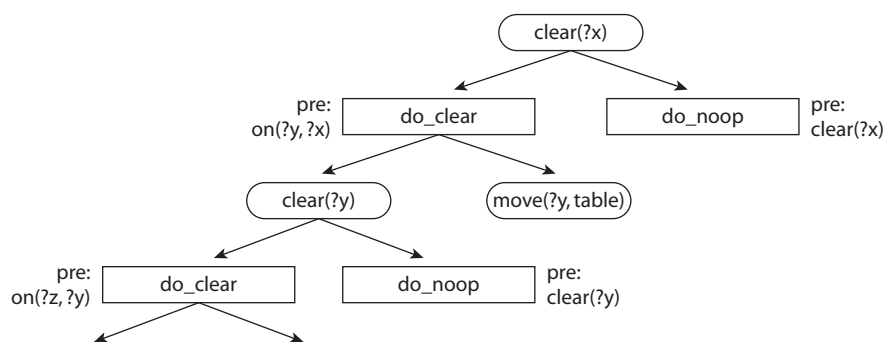


Figure 3.5: Example of a recursive task: the $clear$ task from Blocksworld. Capsules, rectangles and arrows have the same meanings as in Figure 3.3.

Sections 3.3 and 3.4 discuss pMPF detail. This is based closely on CHiPs, except where noted in the text. Sections 3.5 and 3.6 discuss novel approaches used to extend pMPF to produce fMPF, which supports planning in recursive first order domains.

3.3 Propositional plans

The two variants of the new MPF planning mechanism are now described. pMPF is very similar to CHiPs, implementing only the support for task and constraint ownership and new interval

⁵Available from <http://www.cs.bham.ac.uk/~djg>.

based temporal model discussed in the previous section. fMPF is an extension to pMPF that can handle the remaining extensions discussed above, namely support for first order problems and (some types of) recursive tasks and methods. While all of these features have been used in previous planning systems, they have not been applied to task tree and summary information techniques: they are necessary to use task trees and summary information on the “traditional” problems described in Section 1.2.

pMPF is introduced in this section and Section 3.3. As stated above, pMPF is similar to CHiPs apart from a few differences that are noted in the text: the entire mechanism is described here both for completeness and as a grounding for the rest of the thesis. fMPF, meanwhile, is entirely new work and is introduced in Sections 3.5 and 3.6.

3.3.1 Agents and ownership

MPF is only capable of representing problems involving *closed* teams of agents (Section 1.1.4), where the set of agents does not change over time. Each agent has a unique identifier that is used to indicate its responsibility for elements of joint plans.

Task trees are used to represent both joint plans and individual plans. Each node and constraint in the tree is annotated with *owner* information specifying which agents are allowed to change it. *Environmental* nodes and constraints, specified in the initial conditions of the planning problem, are given a virtual “owner” (the name of the *problem* itself): environmental nodes and constraints are known to all agents and can be changed by none.

As is discussed below, ownership of nodes can be sufficiently represented with a single owner per node; a list of zero or more owners is necessary to represent ownership of temporal and variable binding constraints.

3.3.2 World state

World state refers to the external state of the environment. It is this state that agents can change by executing actions from their plans, and it is also this state that dictates when an action can or cannot be performed.

In MPF, the state of the world at any instant is specified as a set of *literals* describing Boolean states that can be manipulated by executives. Examples of state literals include *car_at_home*, *car_at_work*, \neg *bob_at_home* and the other preconditions and effects of the primitive tasks from Figure 3.2.

State constraints The preconditions and effects of tasks are represented as two types of node in the task tree⁶:

Preconditions represent continuous finite time intervals during which a particular state literal must not be changed, for example in the preconditions of a task. This is illustrated in Figure 3.6.

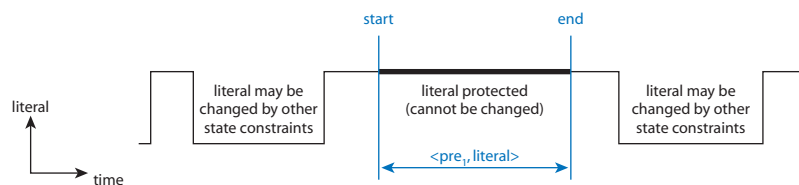


Figure 3.6: Time-line of a precondition state constraint.

Postconditions represent changes to a state literal caused by the planned execution of a task or an initial condition in a problem. In many planning formalisms, postconditions are represented as instantaneous events. However, due to the nature of interval temporal algebra (Section 3.3.4), MPF treats postconditions as short intervals of finite duration. The value of the relevant state literal is undefined during the interval of a postcondition, but is guaranteed to be set correctly from the end of the interval onward. This is illustrated in Figure 3.7.

A state constraint (pre- or postcondition) is a tuple:

$$\langle id, literal_1 \rangle$$

where: *id* is a unique identifier used in temporal constraints and summary conditions to refer to the state constraint and *literal* is the state literal to be constrained or modified.

⁶The complete structure of task trees is discussed in Section 3.3.6 below.

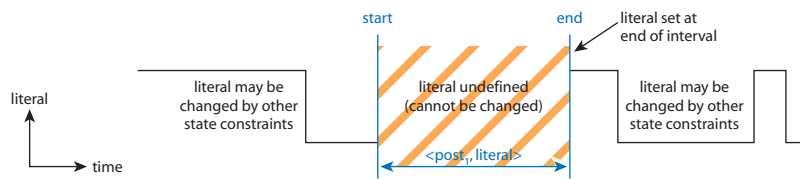


Figure 3.7: Time-line of a postcondition state constraint.

3.3.3 Actions and tasks

In HTN planning, *actions* refer to general types of endeavour, as exemplified by “the robot can open doors”. *Tasks* are specific occurrences of actions within a plan, as exemplified by “the robot opens the door before walking through”. The same action can appear several times in a plan in the form of separate tasks.

Actions in pMPF are represented simply by appropriate propositional identifiers. Examples include *bob_to_work* and *alice_drive_work* from Figure 3.1. Actions are either *primitive* or *abstract*⁷. For a given problem domain, a function `is_primitive(action)` is defined that determines this status for any given action. Primitive actions can be directly executed; abstract actions must be decomposed into smaller parts. For example, the actions *bob_to_work* and *alice_to_shops* in Figure 3.1 are abstract, while all of the actions in Figure 3.2 are all primitive.

Tasks Tasks are particular instances of actions within a plan. In MPF, tasks are a type of node that appears in task trees (Section 3.2.1). Each task occurs over a continuous finite interval during plan execution. A task node is a tuple:

$$\langle id, action, decomps \rangle$$

where: *id* is a unique identifier used in temporal constraints and summary conditions to refer to the task, *action* is the identifier of the action to be performed and *decomps* is a list of *task networks* that can be used to decompose the task (Section 3.3.4). The members of *decomps* are used to describe the possible subplans of abstract tasks and the preconditions and postconditions

⁷These phrases are derived from the standard terms “*abstract task*” and “*primitive task*” from HTN planning, which are described below

of primitive tasks.

Task ownership A function $\text{owner}(task)$ maps each task to the agent responsible for its decomposition and ordering within the plan. This model was chosen for its simplicity, but it has disadvantages. For example, it is impossible in MPF to represent a *joint task* that is owned by two or more agents. Such a feature would require several features of the mechanism and any planning algorithms implemented using it:

- a list would be required to represent the owners of each node in the task tree;
- a policy would be required for assigning ownership to nodes in the subtrees of multiply owned tasks (Section 3.3.6);
- agents would be required to negotiate over who controls the decomposition of shared nodes during planning and the execution of relevant tasks afterwards.

In fact, the single parent, single owner model also prevents agents directly requesting help during planning, as represented by the question: “*Can you help me plan a route through the maze?*” Such requests would require the exchange of subtrees of the task tree during planning (Section 3.3.6), something which is beyond the scope of this work.

3.3.4 Task networks and temporal constraints

A *task network* describes the temporal relationships between a set of tasks, preconditions and postconditions. Task networks are used in conventional HTN planning to represent plans and *methods* for decomposing abstract tasks (Section 2.1.5). In MPF, task networks are also used to store the structure of preconditions and postconditions of primitive tasks. Task networks are another type of node in task trees (Section 3.2.1). This section describes the structure of a task network and the semantics of temporal constraints. The next section describes how task networks are used to represent plans and methods. Section 3.3.6 describes the structure of task trees, including the roles of state constraints, tasks and task networks.

Task networks A task network is a set of tasks, a set of preconditions, and a set of postconditions, all partially ordered by a set of temporal constraints. Task networks are used to represent

both top level plans and task decompositions. A task network is a tuple:

$$\langle id, action, tasks, pre, post, temporal \rangle$$

where: *id* is a unique identifier for the task network; *action* is the action the network is able to decompose (*null* for the root of a task tree); *tasks* is a list of the tasks in the network; *pre* is a list of preconditions to enforce during execution; *post* is a list of postconditions that will occur during execution; *temporal* is a set of temporal constraints on members of $tasks \cup pre \cup post$.

Temporal constraints Temporal constraints are based on the *interval temporal algebra* of Allen (1983). This is a departure from CHiPs, which uses *point temporal algebra* (Dechter et al., 1991). Interval algebra was chosen because it is more flexible than point algebra for representing partial temporal orderings in plans (see below). Temporal constraints are tuples:

$$\langle id_1, id_2, rel \rangle$$

where: id_1 and id_2 are the identifiers of elements of $net \cup children(net)$ and *rel* is a disjunction of possible temporal relationships between them.

Members of *rel* are taken from Allen's list of thirteen *basic temporal relationships*, $\{e, b, a, m, mi, o, oi, d, di, s, si, f, fi\}$, that may exist between two temporal intervals. These relationships are shown graphically in Figure 3.8. *rel* itself represents the set of possible temporal relationships between id_1 and id_2 . For example, the temporal constraint:

$$\langle pat_head, rub_stomach, \{b, a, m, mi\} \rangle$$

means that it is possible to pat one's head before or after rubbing one's stomach (but that the two activities may not overlap).

Discontinuous temporal disjunctions The elements of the example *rel* above form a *discontinuous set* of temporal relationships. A discontinuous set of relationships *R* is defined as a set in which at least one relationship $r_1 \in R$ cannot be continuously changed into one of the

other relationships $r_2 \in R$ by moving the start- and end-points of the intervals involved, without forming an intermediate relationship $r_3 \notin R$.

Consider, for example, the set $R = \{b, o\}$ ⁸. The relationship b (before) cannot be changed into the relationship o (overlaps) by moving the timepoints involved without temporarily forming the intermediate relationship m (meets). Both b and o are in R but m is not, so R is a discontinuous set of relationships.

Such discontinuity is a key advantage of interval algebra over point algebra and other non-disjunctive timepoint based models of similar complexity such as *Simple Temporal Networks (STNs)*. It is often desirable in planning to resolve a conflict between two parts of a plan by ordering one part before or after the other. It may not matter which part comes first: just that they do not overlap. It is impossible to represent disjunctions like this with a simple timepoint based temporal model such as point algebra; the planning algorithm has to branch to investigate both orderings to maintain completeness. With interval algebra, however, such branching is unnecessary: the planner simply creates a single partial plan that represents both sets of orderings and continues from there.

The key disadvantage of interval temporal algebra is that it is purely qualitative: intervals and constraints do not contain metric information about duration and relationships. Discontinuous quantitative disjunctions are possible with disjunctive timepoint based models such as *Disjunctive Temporal Networks (DTNs)*; Dechter et al., 1991), but this comes at the cost of increased computational complexity of constraint propagation and satisfaction.

Updating temporal information The operator \Leftarrow is used in later formulae to denote the addition of a set of temporal constraints to a temporal network and forward propagation using the algorithm described by Allen (1983) to calculate inferred temporal relationships. For example, the line:

$$temporal(net) \Leftarrow \{\langle id_1, id_2, \{b, m\} \rangle, \langle id_2, id_3, \{d, s, f\} \rangle, \dots\}$$

⁸See Figure 3.8 for descriptions of these symbols.

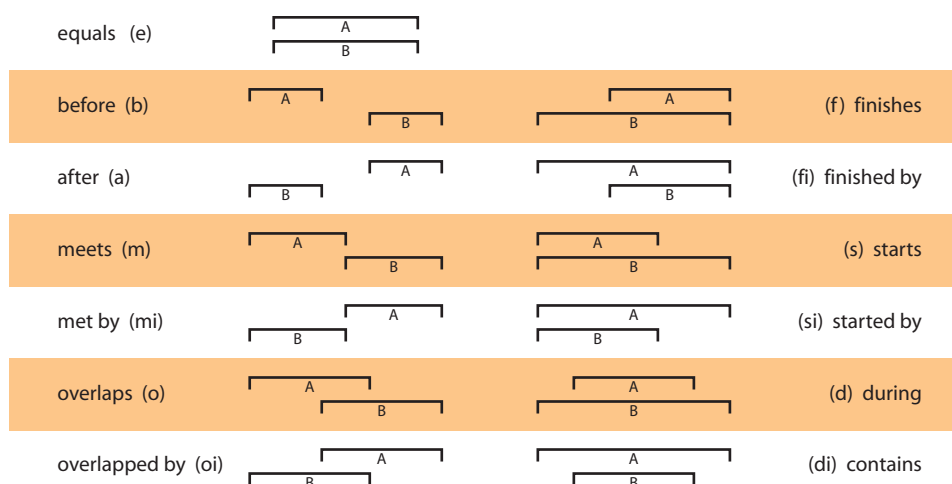


Figure 3.8: Allen's thirteen possible qualitative *basic temporal relationships* between two finite intervals.

represents the addition of a set of constraints to the temporal network in net . After the constraints have been added, the temporal information in the task network is updated to calculate the new inferred relationships between all the relevant temporal intervals. The update algorithm is a simple arc consistency algorithm with $O(n^3)$ complexity, where n is the number of intervals in the temporal network: see Allen (1983) for further details.

The function $tlookup(id_i, id_j, net)$ is used in later formulae to denote the inferred temporal relationship between id_i and id_j according to $temporal(net)$:

$$answer \leftarrow tlookup(id_1, id_2, net)$$

In both cases, id_1 and id_2 may be the identifiers of any of $children(net)$ or of net itself.

Frame constraints To preserve the temporal semantics of abstract tasks in HTN planning⁹, a task network net occupies a temporal interval that exactly contains the intervals of its children. Temporal constraints called *frame constraints* are added to $temporal(net)$ such that for any $node \in children(net)$:

⁹Formalised by Erol (1996).

$$\text{tlookup}(id(node), id(net), net) \subseteq \{e, s, f, d\}$$

meaning that the temporal interval of each $node \in \text{children}(net)$ is equal to, starts, finishes, or is contained by the temporal interval of net , according to the definitions in Figure 3.8.

For convenience, a function `net_timing` is used in later formulae to give the effective timing of a task or state constraint relative to its parent task network:

$$\text{net_timing}(node, net) = \text{tlookup}(id(net), id(node), net)$$

Commutative and transitive temporal operators Two operators are required for reasoning about the relationships between intervals in Allen’s algebra. These are the *commutative temporal operator*, `inv`, which returns the inverse of established relationships, and the *transitive temporal operator*, `trans`, which calculates the relationship between two intervals based upon their relationships with a third interval.

Given a temporal relationship rel of an interval b relative to another interval a , the function $\text{inv}(rel)$ returns the inverse relationship of a relative to b . For example:

$$\begin{aligned} \text{given } rel &= \text{tlookup}(a, b, net) = \{b, m\} \\ \text{then } \text{inv}(rel) &= \text{tlookup}(b, a, net) = \{a, mi\} \end{aligned}$$

Each member of the BTR has an inverse: the inverse of b (“before”) is a (“after”) and so on. The `inv` operator calculates the inverse of a general relationship rel by unioning the inverses of the members of rel . The algorithm for `inv` is shown in Figure 3.9 and uses data from Table 3.1.

```

1  function inv( $rel$ )  $\rightarrow$   $ans$ :
2       $ans \leftarrow \emptyset$ 
3      for each  $r \in rel$ :
4           $ans \leftarrow ans \cup \text{commutation\_table}(r)$ 

```

Figure 3.9: Algorithm for the commutative temporal operator `inv`. The *commutation_table* is shown in Table 3.1.

Given a temporal relationship rel_1 between two intervals a and b , and a relationship rel_2 between

$r_0 \in rel$	e	b	a	d	di	o	oi	m	mi	s	si	f	fi
$r_1 \in inv(rel)$	e	a	b	di	d	oi	o	mi	m	si	s	fi	f

Table 3.1: Inverses of members of the BTR, used in the implementation of the `inv` operator shown in Figure 3.9.

b and a third interval c , the operator $trans(rel_1, rel_2)$ returns the relationship between a and c .

For example:

$$\begin{aligned}
 &\text{given } rel_1 = \text{tlookup}(a, b, net) = \{b, m\} \\
 &\quad \text{and } rel_2 = \text{tlookup}(b, c, net) = \{b, m\} \\
 &\text{then } trans(rel_1, rel_2) = \text{tlookup}(a, c, net) = \{b\}
 \end{aligned}$$

Table 3.2 shows the result of applying `trans` to two atomic temporal relationships, each consisting of a single member of the BTR. The general algorithm for `trans`, shown in Figure 3.10, calculates its result by unioning the tabulated values for each combination of $r_1 \in rel_1$ and $r_2 \in rel_2$.

```

1  function trans( $rel_1, rel_2$ )  $\rightarrow ans$ :
2       $ans \leftarrow \emptyset$ 
3      for each  $r_1 \in rel_1$ :
4          for each  $r_2 \in rel_2$ :
5               $ans \leftarrow ans \cup transition\_table(r_1, r_2)$ 

```

Figure 3.10: Algorithm for the transitive temporal operator `trans`. The `transition_table` is shown in Table 3.2.

Ownership of temporal constraints Whereas tasks have exactly one owner, temporal constraints can be enforced by one or more agents. A function $owners(con)$ maps every constraint con to a (possibly empty) set of agents that wish the constraint to be enforced. Constraints imposed as initial conditions in the definition of the planning problem have an additional “owner” (the name of the problem) to prevent them being retracted by agents during planning.

Constraints with non-empty owners lists are *assumptions* enforced by one or more agents or the environment. Constraints that are inferred from assumptions have no owners.

trans (r_1, r_2)		$r_1 \in rel_1$											
		b	a	d	di	o	oi	m	mi	s	si	f	fi
b	b	any	b	b, di, o, m, fi	b	b, di, o, m, fi	b, di, o, m, fi	b, di, o, m, fi	b	b, di, o, m, fi	b, di, o, m, fi	b	b
a	any	a	a, di, oi, mi, si	a, di, oi, mi, si	a, di, oi, mi, si	a	a, di, oi, mi, si	a, di, oi, mi, si	a	a	a	a	a, di, oi, mi, si
d	b, d, o, m, s	a, d, oi, mi, f	d	$e, d, di, o, oi, s, si, f, fi$	d, o, s	d, oi, f	d, o, s	d, oi, f	d	d, oi, f	d, oi, f	d	d, o, s
di	b	a	any	di	b, di, o, m, fi	a, di, oi, mi, si	b	a	b, di, o, m, fi	di	a, di, oi, mi, si	d, o, s	di
o	b	a, d, oi, mi, f	b, d, o, m, s	di, o, fi	b, o, m	$e, d, di, o, oi, s, si, f, fi$	b	d, oi, f	b, o, m	di, o, fi	d, o, s	d, o, s	o
oi	b, d, o, m, s	a	a, d, oi, mi, f	di, oi, si	$e, d, di, o, oi, s, si, f, fi$	a, oi, mi	b	a	d, oi, f	oi	a, oi, mi	m	di, oi, si
m	b	a, d, oi, mi, f	b	di, o, fi	b	di, o, fi	b	e, s, si	b	di, o, fi	m	m	m
mi	b, d, o, m, s	a	a	di, oi, si	di, oi, si	a	e, f, fi	a	mi	mi	a	di, oi, si	o
s	b	a, d, oi, mi, f	d	di, o, fi	o	d, oi, f	m	d, oi, f	s	e, s, si	d	o	o
si	b	a	a, d, oi, mi, f	di	di, o, fi	a, oi, mi	m	a	e, s, si	si	a, oi, mi	di	di
f	b, d, o, m, s	a	d	di, oi, si	d, o, s	oi	d, o, s	mi	d	oi	f	e, f, fi	e, f, fi
fi	b	a	b, d, o, m, s	di	b, o, m	di, oi, si	b	mi	b, o, m	di	e, f, fi	fi	fi

Table 3.2: Transitive relationships between members of the BTR, used in the implementation of the trans operator shown in Figure 3.10. Trivial transitions involving the e (equal) relationship are not shown.

3.3.5 Problem definitions: initial plans and methods

As described in Section 2.1.5, a single agent HTN planning problem is defined as an *initial plan* containing one or more abstract tasks, and a set of *methods* that map abstract tasks to subplans that achieve them. The planning agent proceeds by iteratively choosing an abstract task in the plan, decomposing it with an appropriate method, and resolving any resulting conflicts, until a conflict free plan is created that contains nothing but primitive tasks.

In MPF, initial plans and methods are represented using task networks. Each method has an *action* field that corresponds to the actions of the tasks it will be used to decompose. If the *action* field is an abstract action, the *tasks* field of the method will describe the subtasks that need to be performed and the *pre* and *post* fields will describe constraints on world state that need to be satisfied before, after, during and between their execution. If the *action* of the method is primitive, the *tasks* field will be empty and the *pre* and *post* fields will describe the preconditions and effects of the action. In an initial plan, the *action* field is *null*, the *tasks* field contains the set of abstract tasks to be performed (the “goals” of the problem), and the *pre* and *post* fields describe initial, intermediate and final conditions. In all cases the *temporal* field is used to describe the temporal relationships between the members of $tasks \cup pre \cup post$.

Multi agent problems are specified by adding ownership information to the tasks and constraints in the initial plan and assigning a set of methods to each agent to act as a *method library*.

Rather than planning by substituting tasks for method bodies according to the conventional HTN paradigm (Section 2.1.5), MPF agents start by building a *task tree* that represents the space of possible decompositions of the tasks in the initial plan, and proceed by pruning branches and rewriting parts of the tree until a final plan is produced. As discussed in Section 3.2, task trees are used as a basis for the calculation of *summary information* about the possible preconditions and effects of abstract tasks, from which useful search heuristics and test functions can be derived.

The next section describes the structure of a task tree, and the generation and maintenance of summary information are described in Sections 3.3.7 to 3.3.9. Section 3.4 goes on to describe summary based heuristics and test functions, and planning operators and refinements used in propositional planning.

3.3.6 Task trees

In MPF, partial plans are represented using structures called *task trees*. A task tree describes a plan and all the possible decompositions of the abstract tasks therein.

Structure of a task tree The nodes of a task tree are of three types: state constraints, tasks and task networks. Each of these types has been described in a previous section. The *root* of the tree (“level 0”) is a task network representing the overall plan. Level 1 nodes are made up of the members of $tasks(root) \cup pre(root) \cup post(root)$, and describe the tasks in the plan and constraints on their execution. The function *children* is defined for convenience to denote the children of a task network *net*:

$$children(net) = tasks(net) \cup pre(net) \cup post(net)$$

Level 2 nodes are made up of members of the *decomps* fields of level 1 tasks, and represent the possible decompositions of those tasks. Level 3 nodes are the children of level 2 task networks, and so on. Task networks are referred to using the following terms depending on their location in the tree:

Plans form the roots of the trees, and have a *null action* field.

Abstract decompositions are children of abstract tasks and have an *action* field matching that of their parent task.

Primitive decompositions are children of primitive tasks. Like abstract tasks they have an *action* field matching that of their parent task. Because primitive tasks are atomic, these networks also have an empty *tasks* field.

A complete task tree for the example in Figure 3.1 has been reproduced in Figure 3.11¹⁰. While the figure is quite complex, it clearly shows the alternating levels of task networks and tasks, and that all leaves of the task tree are state constraints. In each task network, the *temporal* field represents the local temporal relationships between the children of the network.

¹⁰While this textual representation is rather verbose, the various graphical alternatives considered either omitted parts of the tree or were too large to fit on the page.

```

⟨root, null, {
  ⟨task1, bob_to_work, {
    ⟨decomp1, bob_to_work, {
      ⟨task2, bob_get_in_car, {
        ⟨decomp2, bob_get_in_car, {},
          {⟨pre1, car_at_home⟩}, {⟨post1, bob_in_car⟩},
          {⟨pre1, post1, {m}⟩}}}}),
      ⟨task3, bob_drive_work, {
        ⟨decomp3, bob_drive_work, {},
          {⟨pre2, bob_at_home⟩, ⟨pre3, bob_in_car⟩}, {⟨post2, ¬bob_at_home⟩,
            ⟨post3, bob_at_work⟩, ⟨post4, car_at_home⟩, ⟨post5, car_at_work⟩},
          {⟨pre2, pre3, {e}⟩, ⟨pre2, post2, {m}⟩, ⟨post2, post3, {e}⟩,
            ⟨post2, post4, {e}⟩, ⟨post2, post5, {e}⟩}}}}),
      ⟨task4, bob_get_out_car, {
        ⟨decomp4, bob_get_out_car, {},
          {⟨pre4, bob_in_car⟩}, {⟨post6, ¬bob_in_car⟩},
          {⟨pre4, post6, {m}⟩}}}}),
        {⟨task2, task3, {m}⟩, ⟨task3, task4, {m}⟩}},
      ⟨decomp5, bob_to_work, {
        ⟨task5, bob_walk_work, {
          ⟨decomp6, bob_walk_work, {},
            {⟨pre5, bob_at_home⟩}, {⟨post7, ¬bob_at_home⟩, ⟨post8, bob_at_work⟩},
            {⟨pre5, post7, {m}⟩, ⟨post7, post8, {m}⟩}}}}),
          {}}}),
        ⟨task6, alice_to_shops, {
          ⟨decomp7, alice_to_shops, {
            ⟨task7, alice_get_in_car, {
              ⟨decomp8, alice_get_in_car, {},
                {⟨pre6, car_at_home⟩}, {⟨post9, alice_in_car⟩},
                {⟨pre6, post9, {m}⟩}}}}),
              ⟨task8, alice_drive_shops, {
                ⟨decomp9, alice_drive_shops, {},
                  {⟨pre7, alice_at_home⟩, ⟨pre8, alice_in_car⟩}, {⟨post10, ¬alice_at_home⟩,
                    ⟨post11, alice_at_shops⟩, ⟨post12, ¬car_at_home⟩, ⟨post13, car_at_shops⟩},
                  {⟨pre7, pre8, {e}⟩, ⟨pre7, post10, {m}⟩, ⟨post10, post11, {e}⟩,
                    ⟨post10, post12, {e}⟩, ⟨post10, post13, {e}⟩}}}}),
                  ⟨task9, alice_get_out_car, {
                    ⟨decomp10, alice_get_out_car, {},
                      {⟨pre9, alice_in_car⟩}, {⟨post14, ¬alice_in_car⟩},
                      {⟨pre9, post14, {m}⟩}}}}),
                    {⟨task7, task8, {m}⟩, ⟨task8, task9, {m}⟩}}}}),
                    {⟨task1, task6, {m}⟩}}

```

Figure 3.11: Propositional task tree for the problem in Figure 3.1. Task networks and temporal constraints are written in black, tasks in orange, and state constraints in blue.

Generation of an initial task tree Given an initial description of a planning problem consisting of an initial plan *root*, a set of agents *agents* and a set of methods for each agent $methods(agent \in agents)$, it is possible to build a *task tree* using the algorithm in Figure 3.12.

```

1  function build_propositional_task_tree(net):
2      for each t  $\in$  tasks(net):
3          for each m  $\in$  methods(owner(t)):
4              if action(m) = action(t):
5                  add copy of m to decomps(t)
6                  build_propositional_task_tree(m)

```

Figure 3.12: Algorithm for propositional task tree generation, starting with an initial task network *root* and a set of agent specific method libraries $methods(agents)$.

The initial plan is used as the root of the tree. The set of methods that apply to each task in the initial plan is copied and set as the *decomps* field of the task. The algorithm proceeds by recursively adding decomposition task networks to each task in the tree, until a complete tree has been built up. As mentioned above, the leaves of the complete tree are all state constraints.

The single owner, single parent structure of task trees is limiting in situations where agents have heterogeneous planning abilities¹¹. Because agents with different sets of methods may produce different subtrees for a particular task, the reassignment of abstract tasks may involve the recreation of large parts of the task tree and the recalculation of lots of summary information (see below). For this reason, the exchange and reassignment of planning tasks is not considered in this thesis, although it is possible for agents with different sets of methods to plan together in the same multi agent system without exchanging tasks. The “Holes” domain, described in Section 5.2.3, is an example of such a domain.

Recursive tasks The propositional task tree generation algorithm requires that there are no *recursive tasks* (Section 2.1.5) in the initial plan or any of the methods used. A task is recursive if it can be decomposed into a history containing a task with the same action, because the same methods can be applied to the descendant as the ancestor: this would create an infinite recursion at line 6 of the algorithm. More formally, a task *task* is recursive if $is_recursive(task)$, where:

¹¹For example, when agents have different method libraries.

$$\begin{aligned}
& \text{is_recursive}(task) \implies \text{can_decompose}(task, \text{action}(task)) \\
\text{can_decompose}(task, action) & \implies \exists t \in \text{tasks}(m \in \text{methods}(task)) \\
& \quad \text{action}(t) = \text{action}(task) \vee \text{can_decompose}(t, action)
\end{aligned}$$

Task trees form the basic partial plan representation in MPF. Every time a task tree is refined, a set of new trees is generated, and the agent chooses one of them to use as the basis for further refinement. As discussed in Section 3.2.4, agents rely on several heuristics to guide search. These heuristics are based on *summary information* about the possible preconditions and effects of abstract tasks, which is generated directly from the task tree data structure. The next section describes the basic unit of summary information, the *summary condition*. Sections 3.3.8 and 3.3.9 describe the possible relationships between summary conditions and the mechanism for their generation from the task tree. Section 3.4 goes on to describe the information and heuristics that can be generated from summary information, and the planning operators and refinements used in planning.

3.3.7 Summary conditions

Summary conditions (Section 3.2.3) represent the possible preconditions and postconditions of abstract tasks. They represent *possibilities* because abstract tasks can often be decomposed in several ways: the choices of decomposition and temporal ordering will affect the actual preconditions and postconditions present in the final plan. Summary conditions allow the planner to reason directly about potential conflicts at high levels of abstraction without actually decomposing tasks, helping to guide search and minimise backtracking.

Summary information is propagated from the leaves of the task tree upward. A set of functions $\text{sumpre}(node)$, $\text{sumin}(node)$ and $\text{sumpost}(node)$ maps each node in the tree to sets of summary pre-, in- and postconditions.

A summary condition *con* is a tuple:

$$\langle node, \text{existence}, \text{timing}, \text{literal} \rangle$$

where *node* is the identifier of the relevant node in the task tree, *existence* $\in \{must, may\}$ indicates whether the summary condition exists in all or some of the histories of *node*, *timing* represents the timing of *con* relative to *node* and *literal* is the state literal being referred to. *timing* and *existence* are described in more detail below.

Existence of summary conditions The *existence* field is *must* if *con* applies to all of the histories of *node*, and *may* if it only applies to some of them.

Consider, for example, a *task* with two decompositions $decomp_1$ and $decomp_2$ that have the following summary preconditions:

$$\begin{aligned} \text{sumpre}(decomp_1) &= \{\langle must, \{e\}, literal_1 \rangle, \langle must, \{e\}, literal_2 \rangle\} \\ \text{sumpre}(decomp_2) &= \{\langle must, \{e\}, literal_1 \rangle\} \end{aligned}$$

The precondition involving $literal_1$ will be present in the plan in some form no matter which decomposition of *task* is chosen. The precondition involving $literal_2$, on the other hand, will disappear if $decomp_2$ is chosen. To represent this, *task* will inherit summary preconditions as follows:

$$\text{sumpre}(task) = \{\langle must, \{e\}, literal_1 \rangle, \langle may, \{e\}, literal_2 \rangle\}$$

While this example ignores choices of temporal orderings, the existence of summary conditions is dependent on these as much as it is dependent on choices of decomposition. The calculation of the existence of summary conditions is described in detail in Sections 3.3.8 and 3.3.9.

Timing of summary conditions The *timing* field represents the temporal relationship between the interval of $node(con)$ and the interval of *con*. It is equivalent to the *rel* field of the hypothetical temporal relationship $\langle node, con, rel \rangle$. Because each node in a task tree temporally contains all of its children (Section 3.3.4), it also temporally contains all of its summary conditions:

$$timing(con) \subseteq \{e, si, fi, di\}$$

The temporal relationship between any two summary conditions a and b belonging to children of a task network net is denoted in later formulae by the function:

$$\begin{aligned} \text{slookup}(a, b, net) = & \text{trans}(\text{inv}(timing(a)), \\ & \text{trans}(\text{tlookup}(node(a), node(b), net), timing(b))) \end{aligned}$$

where trans , inv and tlookup are as defined in Section 3.3.4.

3.3.8 Achieving, clobbering and undoing

In conventional planning mechanisms, relationships such as achieving, clobbering and undoing are calculated between primitive preconditions and effects. However, equivalent relationships can also be calculated between summary conditions. The difference is that, just as summary conditions have an existence of *must* or *may*, these *summary relationships* can represent relationships that possibly exist between pairs of conditions, as well as relationships that definitely exist. Informally these summary relationships are:

- a postcondition a can *achieve* a precondition b by asserting $literal(b)$ before b begins;
- a postcondition a can *clobber* a precondition b by asserting $\neg literal(b)$ before b ends;
- a postcondition b can *undo* a postcondition a by asserting $\neg literal(a)$ after a ends;
- an incondition a can *clobber* a incondition b by requiring $\neg literal(b)$ during b .

Formally, the relationships are calculated for the children of a task network net as follows. This procedure differs slightly from that in CHiPs because of the new interval temporal algebra introduced in Section 3.3.4:

1. The summary conditions of the children of net are collected into three sets:

$$\begin{aligned}
pres &= \{pre \in \text{sumpres}(child \in \text{children}(net))\} \\
ins &= \{in \in \text{sumins}(child \in \text{children}(net))\} \\
posts &= \{post \in \text{sumposts}(child \in \text{children}(net))\}
\end{aligned}$$

The algorithms used to obtain *sumpres*, *sumins* and *sumposts* are described in the next section.

2. A *relaxed interaction graph* is built where nodes are summary conditions and edges represent possible interactions. This relaxed graph does not take into account *blocking* by intermediate summary conditions. Disjunctions of temporal relationships can make it uncertain whether a pair of summary conditions will interact or not. To cope with this the relaxed graph incorporates two kinds of edges: *must edges* and *may edges*. *May edges* are added wherever an interaction is possible; *must edges* are reserved for situations in which interaction is inevitable¹². Edges are inserted into the relaxed graph for every pair of summary conditions for which the following conditions hold:

$$\begin{aligned}
\text{must_relaxed}(a \in posts, b \in pres) &= \text{literal}(a) \in \{\text{literal}(b), \neg\text{literal}(b)\} \\
&\quad \wedge \text{slookup}(a, b, net) \cap \{m, a\} = \emptyset \wedge \text{must}(a) \\
\text{may_relaxed}(a \in posts, b \in pres) &= \text{literal}(a) \in \{\text{literal}(b), \neg\text{literal}(b)\} \\
&\quad \wedge \text{slookup}(a, b, net) \cup \{m, a\} \supset \{m, a\} \\
\text{must_relaxed}(a \in posts, b \in posts) &= \text{literal}(a) \in \{\text{literal}(b), \neg\text{literal}(b)\} \\
&\quad \wedge \text{slookup}(a, b, net) \cap \{m, a\} = \emptyset \wedge \text{must}(b) \\
\text{may_relaxed}(a \in posts, b \in posts) &= \text{literal}(a) \in \{\text{literal}(b), \neg\text{literal}(b)\} \\
&\quad \wedge \text{slookup}(a, b, net) \cup \{m, a\} \supset \{m, a\} \\
\text{must_relaxed}(a \in ins, b \in ins) &= \text{literal}(a) = \neg\text{literal}(b) \\
&\quad \wedge \text{slookup}(a, b, net) \cap \{b, a, m, mi\} = \emptyset \wedge \text{must}(a) \\
\text{may_relaxed}(a \in ins, b \in ins) &= \text{literal}(a) \in \{\text{literal}(b), \neg\text{literal}(b)\} \\
&\quad \wedge \text{slookup}(a, b, net) \cup \{b, a, m, mi\} \supset \{b, a, m, mi\}
\end{aligned}$$

where the function $\text{must}(con)$ is defined for brevity to indicate the value of $\text{existence}(con)$:

¹²Wherever a *must edge* links two nodes, a *may edge* also links them.

$$\text{must}(con) = \begin{cases} true & \text{if } \text{existence}(con) = \text{must} \\ false & \text{if } \text{existence}(con) = \text{may} \end{cases}$$

Note that the existence value of each edge is independent of the existence of the summary condition being achieved, clobbered or undone. Hence the value of `must_clobbered` is dependent on `must(a)` but not on `must(b)` and the value of `must_undone` is dependent on `must(b)` but not on `must(a)`.

Ignoring optimisations gained by storing preconditions, inconditions and postconditions in separate lists, the complexity of producing the relaxed graph is $O(n^2)$ where $n = |\text{pres} \cup \text{ins} \cup \text{posts}|$.

3. A *constrained interaction graph* is built from the relaxed graph, taking into account *blocking* by intermediate summary conditions:

$$\begin{aligned} \text{must_constrained}(a, b) &= \text{must_relaxed}(a, b) \wedge \neg \text{may_blocked}(a, b) \\ \text{may_constrained}(a, b) &= \text{may_relaxed}(a, b) \wedge \neg \text{must_blocked}(a, b) \end{aligned}$$

Informally, *blocking* occurs when a postcondition a does not affect a precondition b because of an intermediate summary condition c . In practice, because of the disjunctive nature of interval temporal algebra, there are two situations in which this occurs:

Separation when a occurs before c and c occurs before b .

Exclusion when a occurs before or after (but not overlapping or during) an interval env that envelopes c and b .

The *exclusion* relationship is unique to MPF and cannot be represented in CHiPs.

More formally, the `must_blocked` and `may_blocked` relationships are defined as follows:

$$\begin{aligned}
 \text{must_blocked}(a, b) &= \exists c \left(\text{must_separated}(a, c, b) \vee \text{must_excluded}(a, c, b) \right) \wedge \text{must}(c) \\
 \text{may_blocked}(a, b) &= \exists c \left(\text{may_separated}(a, c, b) \vee \text{may_excluded}(a, c, b) \right) \\
 \text{must_separated}(x, y, z) &= \text{must_relaxed}(x, y) \wedge \text{must_relaxed}(y, z) \\
 \text{may_separated}(x, y, z) &= \text{may_relaxed}(x, y) \vee \text{may_relaxed}(y, z) \\
 \text{must_excluded}(x, y, z) &= \exists env \text{ enveloped}(z, y, env) \wedge \text{must_out}(x, env) \\
 \text{may_excluded}(x, y, z) &= \exists env \text{ enveloped}(z, y, env) \wedge \text{may_out}(x, env) \\
 \text{enveloped}(u, v, env) &= \text{must_in}(u, env) \wedge \text{must_in}(v, env) \wedge \text{must_bef}(u, v)
 \end{aligned}$$

and the temporal functions `must_out` and so on are defined as follows:

$$\begin{aligned}
 \text{must_out}(x, y) &= \text{tlookup}(x, y, net) \subseteq \{b, a, m, mi\} \\
 \text{may_out}(x, y) &= \text{tlookup}(x, y, net) \cap \{b, a, m, mi\} \neq \emptyset \\
 \text{must_in}(x, y) &= \text{tlookup}(x, y, net) \subseteq \{e, s, f, d\} \\
 \text{may_in}(x, y) &= \text{tlookup}(x, y, net) \cap \{e, s, f, d\} \neq \emptyset \\
 \text{must_bef}(x, y) &= \text{tlookup}(x, y, net) \subseteq \{b, m\} \\
 \text{may_bef}(x, y) &= \text{tlookup}(x, y, net) \cap \{b, m\} \neq \emptyset
 \end{aligned}$$

where `tlookup` is the temporal lookup function defined in Section 3.3.4.

Any summary condition can block any of the $O(n^2)$ arcs in the relaxed graph, preventing an arc being added to the constrained graph. Thus, given the same definition of n stated above, the complexity of building the constrained graph is $O(n^3)$.

4. The actual *achieves*, *clobbers* and *undoes* relationships for summary conditions of appropriate types can be defined simply in terms of the constrained graph:

$$\begin{aligned}
 \text{must_achieve}(a, b) &= (\text{literal}(a) = \text{literal}(b)) \wedge \text{must_constrained}(a, b) \\
 \text{may_achieve}(a, b) &= (\text{literal}(a) = \text{literal}(b)) \wedge \text{may_constrained}(a, b) \\
 \text{must_clobber}(a, b) &= (\text{literal}(a) = \neg \text{literal}(b)) \wedge \text{must_constrained}(a, b) \\
 \text{may_clobber}(a, b) &= (\text{literal}(a) = \neg \text{literal}(b)) \wedge \text{may_constrained}(a, b) \\
 \text{must_undo}(a, b) &= (\text{literal}(a) = \neg \text{literal}(b)) \wedge \text{must_constrained}(b, a) \\
 \text{may_undo}(a, b) &= (\text{literal}(a) = \neg \text{literal}(b)) \wedge \text{may_constrained}(b, a)
 \end{aligned}$$

As the maximum connectivity of the constrained graph is $O(n^2)$, the complexity of this step is $O(n^2)$.

Effectors, achievers and clobberers If a summary condition of a node $node_1$ achieves a summary condition of a node $node_2$, $node_1$ is called an *achiever* of $node_2$. Similarly, if a summary condition of a node $node_3$ clobbers a summary condition of a node $node_2$, $node_3$ is called a *clobberer* of $node_2$. Achievers and clobberers are collectively known as the *effectors* of a node.

Overall complexity of summarisation The calculation of achieving, clobbering and undoing relationships is central to the calculation of summary information for decomposition task networks and abstract tasks (Section 3.3.9): it is also the complexity defining step in the whole summarisation process. Clement quotes the complexity of calculating this information for a single task network as $O(d^2c^2)$, where d is the number of child nodes and c is the average number of summary conditions per child. This result is equivalent to the complexity of calculating the relaxed graph above because $dc \approx x$. The added complication of taking blockers and exclusion relationships into account raises the equivalent overall complexity of the algorithm above to $O(d^3c^3)$. However, as shall be seen in Section 3.4.3, the added complexity allows for a reduction in the branching factor of the main planning algorithm, which arguably makes it beneficial overall.

Summary relationships have to be calculated once for each decomposition in the initial task tree (Section 3.3.6), and only have to be recalculated during planning when the temporal orderings are changed in the root node in the tree (Section 3.4.3). The overall complexity of initialising the summary information in a task tree of height h is therefore $O(d^3c^3d^h) = O(d^{(h+3)}c^3)$, and the worst case maintenance cost during propositional planning is $O(d^3c^3)$ each refinement¹³.

The d^h term in the initialisation complexity is an unfortunate and inevitable consequence of working with tree structures. In non-recursive problems, h is a small fixed value determined by the relationships between methods and is no cause for concern. In recursive problems (Section 3.6) the initial value of h must be estimated: the *value counting* estimation technique used in this thesis (Section 3.6.1) relates h to the number of objects (blocks, tables, grippers and so on) in the world, so initialisation can become time consuming for larger problems. In these cases, d can be significantly reduced by moving to a first order representation of actions and world state

¹³The situation is much worse in first order planning, as shall be seen in Section 3.5.3.

(Section 3.5). In the empirical comparison of pMPF and fMPF in Section 5.3, initialisation is only prohibitively slow for propositional representations.

3.3.9 Summarising task tree nodes

Summary information is calculated recursively in the task tree, starting at the leaves (state constraints) and working upwards towards the root. The summary conditions of tasks and task networks are calculated in terms of the summary conditions of their children in the tree. Each type of node is summarised using a different algorithm. All three algorithms are described below.

Summarising state constraints State constraints either represent preconditions or postconditions of their grandparent tasks, depending on whether they are in the *pre* or *post* field of the relevant decomposition task network. A precondition *pre* generates a single summary precondition:

```

1 function summarise_pre(net) → (sumpres, sumins, sumposts):
2   sumpres ← {⟨id(pre), must, {e}, literal(pre)⟩}
```

Similarly, a postcondition *post* generates a single summary postcondition:

```

1 function summarise_post(net) → (sumposts, sumins, sumposts):
2   sumposts ← {⟨id(pre), must, {e}, literal(pre)⟩}
```

Summarising task networks The summary conditions of a task network *net* represent the effective pre-, in- and postconditions of all children(*net*). Some information is abstracted away during summarisation: summary conditions with similar literals are merged and temporal information is respecified relative to *net* rather than specific children. First, the *achieves*, *clobbers* and *undoes* relationships are determined between the children of *net* using the procedure described in the previous section. The summary constraints of *net* itself are then calculated from these relationships using the algorithm in Figure 3.13, which is described informally below:

Child summary conditions are considered in groups that have the same literal. For each group:

- A parent summary precondition is output if there is a child summary precondition that is not `must_achieved` or `must_clobbered` by another child summary condition (lines 11, 13). The existence of the parent precondition is *must* if there is a *must* child precondition that is definitely unaffected by other conditions: it is *may* otherwise.
- A parent summary incondition is output if there is a child summary precondition that is `may_achieved` or `may_clobbered` by another child summary condition (lines 8, 10). The existence of the parent incondition is *must* if all child preconditions are definitely affected by other conditions: it is *may* otherwise.
- A parent summary postcondition is output if there is a child summary postcondition that is not `must_undone` by another child summary condition (lines 23, 25). The existence of the parent postcondition is *must* if there is a *must* child postcondition that is definitely unaffected by other conditions: it is *may* otherwise.
- A parent summary incondition is output if there is a child summary postcondition that is `may_undone` by another child summary condition (lines 20, 22). The existence of the parent incondition is *must* if all child postconditions are definitely affected by other conditions: it is *may* otherwise.
- A parent summary incondition is output if there are any child summary inconditions (line 16). The existence of the parent incondition is *must* if any child incondition has an existence of *must*: it is *may* otherwise.

In each case, the timing of the parent summary conditions is the union of the timings of equivalent child summary conditions (line 29).

The complexity of summarising a task network is determined by the complexity of calculating achieves, clobbers and undoes relationships between its children ($O(d^3c^3)$, where d is the number of child nodes and c is the average number of summary conditions per node; Section 3.3.8). The resulting summary information can be calculated directly from the final interaction graph in $O(n^2)$ time.

```

1  function summarise_net(net) → (sumpres, sumins, sumposts):
2      id ← id(net)
3      collect (pres, ins, posts) from summarise_task(task ∈ tasks(net)) ∪
          summarise_pre(pre ∈ pre(net)) ∪ summarise_post(post ∈ post(net))
4      calc relaxed and constrained interaction graphs from (pres, ins, posts)
5      for each pre ∈ pres:
6          timing' ← trans(net_timing(node), timing(pre))
7          if ∃x must_achieve(x, pre):
8              sumins ← merge(⟨id, existence(pre), timing', literal(pre)⟩, sumins)
9          else if ∃x may_achieve(x, pre):
10             sumins ← merge(⟨id, may, timing', literal(pre)⟩, sumins)
11             sumpres ← merge(⟨may, timing', literal(pre)⟩, sumpres)
12          else:
13             sumpres ← merge(⟨id, existence(pre), timing', literal(pre)⟩, sumpres)
14      for each in ∈ ins:
15          timing' ← trans(net_timing(node), timing(in))
16          sumins ← merge(⟨id, existence(in), timing', literal(in)⟩, sumins)
17      for each post ∈ posts:
18          timing' ← trans(net_timing(node), timing(post))
19          if ∃x must_undo(x, post):
20             sumins ← merge(⟨id, existence(post), timing', literal(post)⟩, sumins)
21          else if ∃x may_undo(x, post):
22             sumins ← merge(⟨id, may, timing', literal(post)⟩, sumins)
23             sumposts ← merge(⟨id, may, timing', literal(post)⟩, sumposts)
24          else:
25             sumposts ← merge(⟨id, existence(post), timing', literal(post)⟩, sumposts)

26 function merge(con, set) → set':
27     if ∃con' ∈ set literal(con) = literal(con'):
28         existence' ← existence(con) ∧ existence(con')
29         timing' ← timing(con) ∪ timing(con')
30         set' ← (set − con') ∪ {⟨id, existence', timing', literal(con)⟩}
31     else:
32         set' ← set ∪ {con}

```

Figure 3.13: Algorithm for summarising a task network. The \wedge symbol represents the conjunction of existence values: $e_1 \wedge e_2$ is *must* if e_1 and e_2 are both *must*: it is *may* otherwise.

Summarising tasks The summary conditions of a task $task$ represent the possible choices of summary conditions from $decomps(task)$. A task $task$ is summarised using the algorithm in Figure 3.14. The literals from all summary pre-, in- and postconditions of children of $decomps(task)$ are collected together into three sets: $pre_literals$, $in_literals$ and $post_literals$. Then, for each literal in the three sets, a separate set $cons$ is built containing all matching child summary conditions from which a single final summary condition is output:

$$\langle id(task), \bigvee_{con \in cons} existence(con), \bigcup_{con \in cons} timing(con), literal \rangle$$

where \bigvee represents a disjunction of existence values:

$$\bigvee_1^n e_i = \begin{cases} must & \text{if all } e_i \text{ are } must \\ may & \text{if any } e_i \text{ is } may \end{cases}$$

A *must* summary condition will only be output if there is an equivalent *must* condition for every task network in $decomps(tasks)$, meaning the summary condition will definitely hold no matter which decomposition is chosen. As above, the *timing* of the parent summary condition is the union of the timings of child conditions.

```

1  function summarise_task(task) → (sumpres, sumins, sumposts):
2      id ← id(task)
3      collect (tempres, tempins, temposts) from summarise_net(net ∈ decomps(task))
4      pre_literals ← literal(pre ∈ tempres)
5      in_literals ← literal(in ∈ tempins)
6      post_literals ← literal(post ∈ temposts)
7      for each literal ∈ pre_literals:
8          cons ← {con ∈ tempres | literal(con) = literal}
9          add ⟨id, ∨ existence(con ∈ cons), ∪ timing(con ∈ cons), literal⟩ to sumpres
10     for each literal ∈ in_literals:
11         cons ← {con ∈ tempins | literal(con) = literal}
12         add ⟨id, ∨ existence(con ∈ cons), ∪ timing(con ∈ cons), literal⟩ to sumins
13     for each literal ∈ post_literals:
14         cons ← {con ∈ temposts | literal(con) = literal}
15         add ⟨id, ∨ existence(con ∈ cons), ∪ timing(con ∈ cons), literal⟩ to sumposts

```

Figure 3.14: Algorithm for summarising a task.

Ignoring the optimising step of hashing summary conditions by literal, the complexity of this

algorithm is $O(n^2)$ where n is the total number of summary conditions of child nodes.

3.4 Propositional planning

Given the task tree based plan representation and summary information described in the previous section, the remainder of the pMPF planning mechanism can now be defined, including: planning operators, refinements, solution and failure test functions, heuristics and the strategy for refinement selection.

Planning in MPF involves removing *flaws* from the initial task tree until a flawless solution tree is found. Flaws are either abstract tasks that require decomposition or *threats* (may_clobber relationships) that require resolution between level 1 summary conditions. Each refinement constitutes the removal of a single flaw. Section 3.4.1 defines the solution and failure test functions in terms of summary information, Section 3.4.2 describes the planning operators used to produce new task trees during planning, Section 3.4.3 describes the refinements used to resolve flaws, and Sections 3.4.4 and 3.4.5 describe other routine maintenance tasks that are required during planning.

3.4.1 Abstract detection of solutions and failures

Consistent versus inconsistent plans It is important to specify the difference between plans that are *failures* and plans that are *inconsistent*. *Consistency* means freedom from contradictions. Inconsistent plans are over-constrained, leaving no possible choices for some decompositions or orderings. A task network *net* is *consistent* if it has a consistent set of temporal constraints and each member of $tasks(net)$ has one or more consistent decompositions:

$$\begin{aligned} \text{is_consistent}(net) = & \forall node, node' \in \text{children}(net) \text{ tlookup}(node, node', net) \neq \emptyset \\ & \wedge \forall task \in tasks(net) |decomps(task)| \geq 1 \\ & \wedge \forall task \in tasks(net) \exists decomp \in decomps(task) \text{ is_consistent}(decomp) \end{aligned}$$

Agents often produce inconsistencies during planning. If inconsistencies are introduced in de-

composition task networks, their subtrees are pruned from the task tree (Section 3.4.4). If the root network is made inconsistent the plan is immediately discarded, preventing it being used for further refinement or communication to other agents.

Solutions and failures Solutions and failures are both types of consistent plan: failures represent consistent plans that do not achieve the goals set out in the planning problem or that contain unresolvable conflicts between state constraints. It is difficult to detect in general whether a task tree represents an abstract solution or failure without exhaustively searching through all of its histories. However, summary information can be used as the basis of two qualitative heuristics to identify some such trees without a complete search:

The can_any_way heuristic A task network net is an *abstract solution* if it has no threats (may_clobber or must_clobber relationships) between the summary conditions of the members of $children(net)$:

$$\begin{aligned} \text{can_any_way}(net) &= \text{is_consistent}(net) \wedge \\ &\quad \forall con_1, con_2 \in \text{all_sums}(net) \neg \text{may_clobber}(con_1, con_2) \\ \text{all_sums}(net) &= \text{sumpre}(node \in \text{children}(net)) \cup \text{sumin}(node \in \text{children}(net)) \\ &\quad \cup \text{sumpost}(node \in \text{children}(net)) \end{aligned}$$

In other words, it does not matter which decompositions or groundings are chosen during planning: the plan will always achieve the agent's goals. `can_any_way` is used as the solution detection function in the refinement planning algorithm.

The might_some_way heuristic A task network net is an *abstract failure* if it contains one or more *unresolvable* threats. An unresolvable threat is a `must_clobber` relationship where the clobbered summary condition cannot be removed from the plan by decomposition:

$$\begin{aligned} \text{might_some_way}(net) &= \text{is_consistent}(net) \wedge \\ &\quad \forall con_2 \in \text{all_sums}(net) (\text{existence}(con_2) = 1) \implies \\ &\quad \nexists con_1 \in \text{all_sums}(net) \text{ must_clobber}(con_1, con_2) \end{aligned}$$

In other words, it does not matter which decompositions or groundings are chosen during planning: the plan will never achieve the agent's goals. `might_some_way` is used to encourage early backtracking and prune branches of search. Agents use individual or joint definitions of these functions in different circumstances. Planning algorithms are discussed in detail in the next chapter.

Even though `can_any_way` and `might_some_way` are heuristic functions, they can be used to test for solutions and failures without loss of soundness or completeness because they are both conservative about the solutions and failures they recognise. The functions only return true in situations when the plan is definitely a solution or failure: some solution and failure plans are not detected, and the functions always return false when plans are not solutions or failures. The functions are proven to be correct for total orderings of tasks but not for partial orderings (Appendix D of Clement, 2002). As the refinements and operators below always tend towards total orderings, one of the two functions will always return true eventually, allowing the agent to return a solution plan or backtrack accordingly.

Once the achieves, clobbers and undoes relationships are known between the children of a task network net , `can_any_way(net)` and `might_some_way(net)` can be calculated in $O(n^2)$ time. Again, n is the number of summary conditions of the children of net (Section 3.3.8).

3.4.2 Planning operators

The planning algorithms presented in Chapter 4 use a common set of refinements to remove flaws from task trees. These refinements, which are discussed in the next section, are implemented using the planning operators described below.

Many of the planning operators in this section change the task tree such that some summary information needs to be recalculated: any change to the children or temporal constraints of a node invalidates its summary conditions the conditions of its ancestor nodes. A *dirty* flag is set in appropriate nodes after a planning operator has been applied to indicate that its summary information must be recalculated. Recalculation is performed lazily to minimise the computation necessary to maintain the trees.

Selecting and blocking decompositions A task $task$ must have a single decomposition in $decomps(task)$ before it can be decomposed. If the task has more than one decomposition, a single $decomp \in decomps(task)$ must be *selected* as shown in Figure 3.15. Conversely, if the subtree of a single $decomp$ is inconsistent it must be deleted from the task tree. This process, shown in Figure 3.16, is called *blocking*.

- 1 **function** $select_op(task, decomp)$:
- 2 delete all $decomps(task)$ except $decomp$
- 3 set *dirty* flag for $task$ and all its ancestors

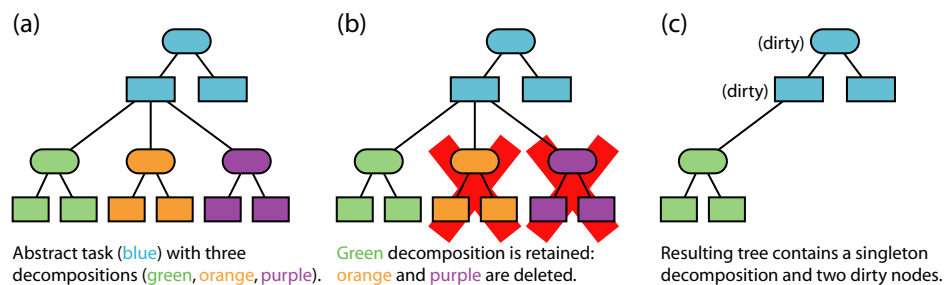


Figure 3.15: Algorithm for, and example of, the propositional $select_op$ planning operator.

- 1 **function** $block_op(task, decomp)$:
- 2 delete $decomp$ from $decomps(task)$
- 3 set *dirty* flag for $task$ and all its ancestors

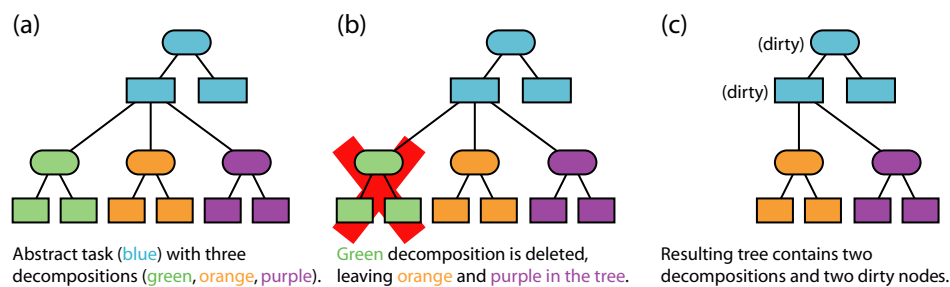


Figure 3.16: Algorithm for, and example of, the propositional $block_op$ planning operator.

Decomposing tasks An abstract task $task$ in a plan $plan$ can be decomposed directly into its component parts if there is only one decomposition in $decomps(task)$. The procedure is shown in Figure 3.17. Note that the temporal interval for $task$ is left in $temporal(net)$ after decomposition. Line 7 of the algorithm adds temporal constraints from the selected decomposition $decomp$ to $temporal(net)$. Existing temporal relationships between $task$ and its siblings

are maintained through the frame constraints from $temporal(decomp)$ and the extra constraint added in line 5.

```

1  function decompose_op(net, task):
2      delete task from tasks(net)
3      for each node ∈ children(decomp ∈ decomps(task)):
4          add child to children(net)
5           $temporal(net) \leftarrow \{ \langle id(task), id(decomp), \{e\} \rangle \}$ 
6      for each con ∈  $temporal(decomp \in decomps(task))$ :
7           $temporal(net) \leftarrow \{con\}$ 
8      set dirty flag in net and all its ancestors

```

Example:

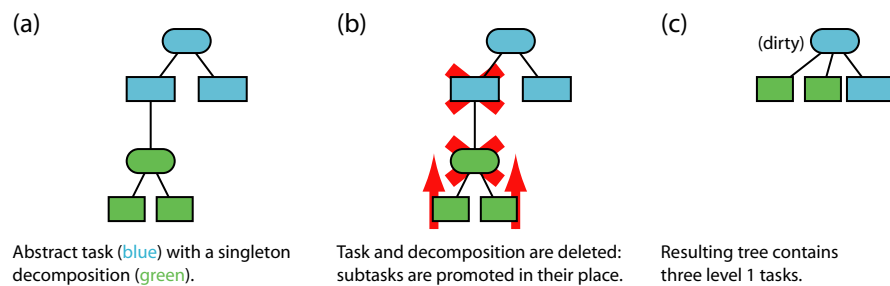


Figure 3.17: Algorithm for, and example of, the propositional decompose_op planning operator.

Ordering and adding envelopes A pair of nodes $node_1$ and $node_2$ in a plan $plan$ can be ordered by simply adding a temporal constraint to $temporal(plan)$. It is sometimes also useful to add extra temporal intervals to a temporal network to act as temporal *envelopes* containing pairs of tasks (Section 3.4.3). Envelopes allow the agent to take advantage of the novel *separation* and *exclusion* relationships described in Section 3.3.8 to remove clobberers by temporal ordering without branching search. This is described in detail with the `order_ref` refinement in the next section.

Pseudocode for the `order_op` and `add_env_op` operators is shown in Figure 3.18. Note that while the `add_env_op` operator does not require the setting of any *dirty* flags, it is only ever used in conjunction with `order_op`. These operators are discussed in more detail in Section 3.4.3.

```

1  function order_op(net, task1, task2, rel):
2      temporal(net)  $\leftarrow$  {id(task1), id(task2), rel}
3      set dirty flag for net and all its ancestors
4  function add_env_op(net, env):
5      add identifier env to temporal(net)

```

Figure 3.18: Algorithm for the `order_op` and `add_env_op` planning operators.

3.4.3 Plan refinements

Each *refinement* in pMPF focuses on a single “flaw” and generates a new plan for each way of resolving that flaw. A flaw may be an abstract task that needs decomposing or a threat between summary conditions that needs resolution.

These refinements are original aspects of MPF: Clement (2002) uses refinements that concentrate on single choices of decomposition or temporal ordering. While the `decompose_ref` refinement described below is very similar to the decomposition refinement used in CHiPs, the `order_ref` refinement is different as it deals with specific sets of threats, and may add many temporal constraints in a single refinement. `order_ref` takes advantage of temporal envelopes (Section 3.4.2) and the exclusion relationship (Section 3.3.8) made possible by interval temporal algebra (Section 3.3.4; Allen, 1983). Flaw oriented refinements are chosen because, when needed, they allow the direct removal of specific threats in or between agents’ plans.

Strategy for refinement choice Agents try to identify the most constraining part of their plan for refinement. This is done on a *per node* basis within level 1 of the task tree. The number of threats on each node is counted and the node with the most threats is refined next. One of the following refinements is chosen, depending on whether the node is a state constraint, abstract or primitive task.

Decompose refinement *Abstract tasks* can be decomposed¹⁴ by replacing them with the children of one of their decompositions. *Primitive tasks* with multiple decompositions can be reduced to a single decomposition to provide a definite set of preconditions and effects. When

¹⁴Clement uses the term “expand” to refer to the decomposition of an abstract task using a method. Erol (1996) uses the term “reduce” to refer to the same thing. As these terms are somewhat conflicting, this thesis opts for neither and uses the term “decompose”, which is also popular in the HTN planning literature.

applied to a task $task$ in a plan $plan$, the `decompose_ref` refinement outputs a new plan for each decomposition in $decomps(task)$, as shown in Figure 3.19. Although the refinement itself does not decompose $task$, the singleton child created will trigger decomposition in the *simplification* post-processing step (Section 3.4.4).

```

1  function decompose_ref( $plan, task$ )  $\rightarrow$   $newplans$ :
2      for each  $decomp \in decomps(task)$ :
3          create a copy  $newplan$  of  $plan$ 
4          locate equivalent  $newtask$  and  $newdecomp$  for  $task$  and  $decomp$ 
5          select_op( $newtask, newdecomp$ )
6          add  $newplan$  to  $newplans$ 

```

Figure 3.19: Algorithm for the propositional `decompose_ref` refinement.

`decompose_ref` is applied to abstract and primitive tasks that have more than one decomposition when they are selected for refinement.

Order refinement Once a task has been fully decomposed, further threats may have to be removed by adding ordering constraints to the plan. For a task $task$ in a plan $plan$ this is done with the algorithm shown in Figure 3.22, described informally as follows:

1. A set *clobberers* is made containing nodes that may clobber one or more of the summary preconditions of $task$.
2. A set *achievers* is made containing nodes that must achieve one or more of the summary preconditions of $task$, and are not in *clobberers*.
3. For each achiever ach a separate plan is output in which an envelope interval env is added to $temporal(plan)$ such that env contains $task$ (Figure 3.20):

$$temporal(plan) \Leftarrow \{\langle env, task, \{di\} \rangle\}$$

ach is ordered inside env and before $task$:

$$temporal(plan) \Leftarrow \{\langle env, ach, \{di\} \rangle, \langle ach, task, \{b, m\} \rangle\}$$

and all *clobberers* are ordered before or after *env*:

$$temporal(plan) \Leftarrow \{ \langle env, clo, \{b, a, m, mi\} \rangle \mid clo \in clobberers \}$$

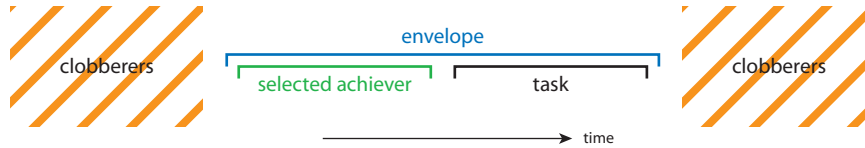


Figure 3.20: Temporal relationships created by `order_ref` when an achiever is selected.

Notice that the ordering in Figure 3.20 implements the *exclusion* arrangement described in Section 3.3.8. It is here that interval temporal algebra provides its flexibility: the ability to producing a single plan in which clobberers are ordered before *or* after a temporal envelope is significant as it drastically reduces the number of plans output by `order_ref`. Point algebra would require one plan to be output for each combination of before and after orderings for each potential clobberer.

4. An extra plan is output in which no achiever is used. Instead, all *clobberers* are simply ordered after *env* (Figure 3.21):

$$temporal(plan) \Leftarrow \{ \langle env, clo, \{a, mi\} \rangle \mid clo \in clobberers \}$$

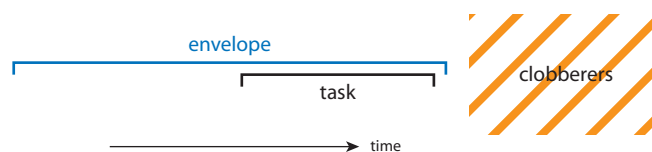


Figure 3.21: Temporal relationships created by `order_ref` when no achiever is selected.

5. Sometimes it is impossible to resolve all threats on a task without first decomposing some of its effectors. To ensure completeness, extra plans are generated by applying `decompose_ref` to the most constraining task from $achievers \cup clobberers$ that has more than one decomposition.

```

1  function order_ref(plan, task) → newplans:
2      for each con in sumpre(task) ∪ sumin(task):
3          achievers ← achievers ∪ {ach | must_achieve(ach, con)}
4          clobberers ← clobberers ∪ {clo | may_clobber(clo, con)}
5      achievers ← achievers − clobberers
6      for each ach ∈ achievers:
7          newplan ← copy(plan)
8          generate a unique temporal interval identifier env
9          add_env_op(plan, env)
10         order_op(plan, env, id(task), {e, si, fi, di})
11         order_op(plan, env, id(ach), {e, si, fi, di})
12         order_op(plan, id(task), id(ach), {b, m})
13         for each clo ∈ clobberers:
14             order_op(plan, env, id(clo), {b, a, m, mi})
15         if is_consistent(newplan):
16             newplans ← newplans ∪ newplan
17     newplan ← copy(plan)
18     for each clo ∈ clobberers:
19         order_op(plan, id(task), id(clo), {b, m})
20     if is_consistent(newplan):
21         newplans ← newplans ∪ newplan
22     choose eff that appears most often in (node(con ∈ achievers ∪ clobberers))
23     newplans ← newplans ∪ decompose_ref(plan, eff)

```

Figure 3.22: Algorithm for the propositional order_ref refinement.

While this technique is not guaranteed to remove all threats to *task*, it will always remove or decompose at least one threat or task. Subsequent uses of order_ref will remove further threats until the task is threat free.

order_ref is applied to state constraints and “singleton” primitive tasks that have one decomposition when they are selected for refinement.

3.4.4 Simplifying task trees

The refinements described above can generate task trees with inconsistent decomposition task networks and singleton abstract tasks that require pruning. These operations are performed as a post-processing step called *simplification*, the result of which is either an inconsistent plan that can be discarded or a consistent plan that can be used as a basis for further refinement. Simplification comprises a number of operations:

Pruning Decompositions with inconsistent subtrees must be blocked from the task tree (Section 3.4.2). As described in Section 3.4.1, inconsistencies can arise from over-constrained temporal or binding constraints, or descendant tasks with no remaining child decompositions.

Decomposition Tasks with singleton decompositions after pruning must be decomposed as described in Section 3.4.2.

Simplification is illustrated in Figure 3.23. Frame (a) shows a task tree with two **blue** level 1 tasks. The left hand task has two decompositions: the **green** decomposition represents a valid task network, and the **orange** decomposition represents a task network with invalid temporal constraints. The invalid task network is pruned from the tree using `block_op` (b). After pruning, the **green** decomposition is a singleton: its parent task is decomposed (c) and replaced with its children (d).

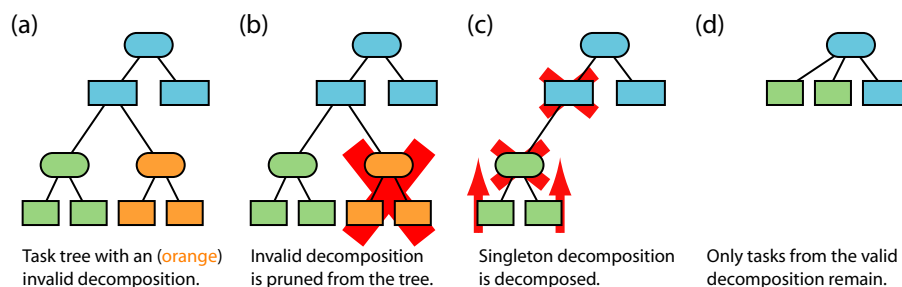


Figure 3.23: Example of simplification.

Pruning can trigger decomposition when it leaves singleton task networks in the task tree. Similarly, decomposition can trigger pruning when it introduces temporal inconsistencies in the parent task network (the **blue** task network in Figure 3.23). Because of this mutual recursion, simplification always results in either a valid task tree, or in a tree with an inconsistent root task network. According to the definitions in Section 3.4.1, trees with inconsistent roots represent inconsistent plans and are dropped from consideration.

Simplification is done in a depth first traversal of the task tree, starting at the leaves and working towards the root. The algorithm is shown in Figure 3.24.

```

1  function simplify(node):
2      if node is a task network:
3          for each child ∈ tasks(node):
4              simplify(child)
5              if |decomps(child)| = 1:
6                  decompose_op(node, child)
7      else:
8          for each child ∈ decomps(node):
9              if is_consistent(child):
10                 simplify(child)
11             else:
12                 block_op(node, child)

```

Figure 3.24: Algorithm for simplifying a propositional task tree.

3.4.5 Splitting and merging plans

As discussed in Section 3.1, task trees can be used to represent *individual plans* comprising tasks and state constraints belonging to a single agent, or *joint plans* comprising tasks and state constraints from multiple agents.

A joint plan p_{joint} in a problem $problem$ can be split into individual plans by creating a plan p_{agent} for each $agent$ and copying into it the subtree for every level 1 $node$ for which:

$$\text{owner}(node) \in \{agent, problem\}$$

and every temporal constraint con for which:

$$\text{owners}(con) \cap \{agent, problem\} \neq \emptyset$$

Temporal constraints are copied across even if they refer to one or more nodes that are not in p_{agent} : these form *inter-plan* constraints that act as coordination information for the individual plans. Because every task and assumption in the joint plan is owned by at least one agent (or the problem), and all assumptions are copied into one or more individual plans, the set of individual plans can be subsequently merged back into a joint plan with no loss of information.

3.5 First order plans and planning

Sections 3.3 and 3.4 described the pMPF planning mechanism, which dealt exclusively with non-recursive propositional planning problems. However, many “traditional” HTN planning problems have first order actions and world state. This section describes the extensions required to pMPF to allow it to handle such problems. A new planning formalism, fMPF, is created, which uses first order task trees as its plan representation. Section 3.6 discusses methods for applying fMPF to recursive planning problems, which completes the development of the planning mechanisms used in this thesis. The extensions described in this section and Section 3.6 are novel contributions to task tree and summary based information techniques.

While propositional plan representations are suitable for solving small problems, they quickly scale up and become unwieldy when modelling problems with multiple instances of objects in them such as Blocksworld, general route navigation and many scheduling problems. This is because an extra branching factor is incurred in the task tree whenever an action can be applied to more than one object in the problem. This limitation can be partly overcome by adding first order state literals and actions to pMPF to produce a new mechanism, fMPF. This section discusses the additional features of this new mechanism.

3.5.1 First order plans

In first order problems the world is made up of a collection of *objects*. State literals and actions become first order predicates defined on tuples of objects. The semantics are the same as those for objects in STRIPS (Section 2.1.2). Objects have an infinite lifetime; they cannot be created or destroyed. Plans can only affect them by altering the values of the literals describing their properties and relationships. Examples of objects may include blocks, surfaces and grippers in Blocksworld, locations and vehicles in logistics domains, and discs and pins in Towers of Hanoi. However, the domain designer has some freedom when it comes to modelling aspects of the problem using objects or state literals.

Planning variables and binding constraints Planning variables represent disjunctions of possible objects in an action or state literal in a plan. For example, the action $move(Birmingham, ?des)$ describes movement from Birmingham to an undecided location $?des$. A list of possible values of $?des$ is stored as part of the plan. Variables are related to one another and to individual world objects by *binding constraints*. There are two types of constraints, *equal* constraints and *not equal* constraints, written as follows:

$$\langle id_1, id_2, eq \rangle$$

$$\langle id_1, id_2, ne \rangle$$

where id_1 and id_2 are identifiers of planning variables or world objects. The binding constraints on a variable affect the set of world objects that form its domain of possible values.

Types of objects Objects, planning variables and arguments in literals and actions may be given *types*. This prevents the agent considering, for example, a gripper as a possible value of a variable representing a block in Blocksworld. The implementation of a flexible type hierarchy is straightforward as objects and types cannot be created or destroyed during planning.

fMPF uses a type hierarchy in which each type has a set of possible values and a set of parent types. When the initial task tree is generated for the planning problem, each planning variable is assigned an initial domain based on the domain of its type and its descendant types. An example type hierarchy is shown in Figure 3.25, together with initial values of variables of various types.

$$\begin{array}{ll} \mathit{supertypes}(!staff) = \{\} & \mathit{values}(!staff) = \{alice, bob\} \\ \mathit{supertypes}(!student) = \{\} & \mathit{values}(!student) = \{charlie, dennis\} \\ \mathit{supertypes}(!phd) = \{!staff, !student\} & \mathit{values}(!student) = \{ellie\} \\ \mathit{type}(?x) = !staff & \mathit{initial_values}(?x) = \{alice, bob, ellie\} \\ \mathit{type}(?y) = !student & \mathit{initial_values}(?y) = \{charlie, dennis, ellie\} \\ \mathit{type}(?z) = !phd & \mathit{initial_values}(?z) = \{ellie\} \end{array}$$

Figure 3.25: Example type hierarchy and planning variables. Type names are prefixed with exclamation marks; variable names are prefixed with question marks.

State literals and actions are defined with applicable type signatures to make sure variables of the correct types are used in the correct places. However, this information is unnecessary once

a valid task tree has been set up with appropriate sets of initial values for each variable.

Type information is ignored in the remainder of this chapter and Chapter 4 as it does not affect the discussion of planning mechanisms or algorithms. Type information is available in Appendix B for each of the planning domains used in the empirical analysis in Chapter 5.

Task networks Binding constraints and value lists for planning variables are stored in new structures inside task networks:

$$\langle id, action, tasks, pre, post, temporal, vars, bindings \rangle$$

where *vars* is a map of variable identifiers declared in this network to sets of their possible values, and *bindings* is a set of constraints on pairs of variables.

The operator \Leftarrow is used to represent the addition of a set of binding constraints to a task network and the calculation of inferred constraints and possible values of variables:

$$bindings(net) \Leftarrow \{ \langle id_1, id_2, eq \rangle, \langle id_2, id_3, ne \rangle, \dots \}$$

The update algorithm used is a variant of the temporal update algorithm (Section 3.3.4), also with $O(n^3)$ complexity where n is the number of variables in the task network. In later formulae, the function $vlookup(id_1, id_2, net)$ returns one of the symbols *eq*, *ne* or *any* representing the inferred relationship between the variables or values id_1 and id_2 . The function $values(?var, net)$ determines the possible values for a variable according to $vars(net)$.

Free and bound variables In decomposition task networks, variables are classified as *bound* or *free* depending on whether they appear in $action(net)$. Bound variables appear in *action* and are constrained by the values of the variables of the corresponding *action* in the parent task. By contrast, free variables can be assigned values arbitrarily within the confines of constraints in $bindings(net)$.

Unification It is possible for partially lifted actions and state literals to share possible groundings even if they contain different planning variables. The unify function shown in Figure 3.26 is used to check whether pairs of actions or state literals share common groundings. In actual fact, this is a slight misnomer as the function does more than conventional unification. In addition to checking whether there is a set of substitutions that can rewrite one action or literal as another, it also checks the values of and binding constraints on relevant planning variables. The function returns *false* regardless of normal unification if actions or literals do not share any common groundings.

```

1  function unify( $a, b, net$ )  $\rightarrow ans$ :
2      if  $a$  and  $b$  have different function names or numbers of arguments:
3           $ans \leftarrow false$ 
4      else:
5          for each  $a_i, b_i$  in the arguments of  $a$  and  $b$ :
6              if  $a_i$  is a planning variable:
7                  if  $b_i$  is a planning variable:
8                      if  $values(a_i, net) \cap values(b_i, net) \neq \emptyset \vee$ 
9                           $vlookup(a_i, b_i, net) = ne$ :
10                          $ans \leftarrow ans \cup a_i/b_i$ 
11                     else:
12                         return  $ans \leftarrow false$ 
13                 else:
14                     if  $b_i \in values(a_i, net)$ :
15                          $ans \leftarrow ans \cup a_i/b_i$ 
16                     else:
17                         return  $ans \leftarrow false$ 
18                 else:
19                     if  $b_i$  is a planning variable:
20                         if  $a_i \in values(a_i, net)$ :
21                              $ans \leftarrow ans \cup a_i/b_i$ 
22                         else:
23                             return  $ans \leftarrow false$ 
24                     else:
25                         if  $a_i \neq b_i$ :
26                             return  $ans \leftarrow false$ 

```

Figure 3.26: Algorithm for “unification” in first order problems. x/y denotes a pair representing the substitution of x for y in formula a or y for x in formula b . The *return* keyword stops execution and returns from the function: otherwise, execution ends naturally with the current value of *ans* as the result.

3.5.2 Approaches to handling first order problems

There are two basic approaches to handling first order planning problems: compilation to a propositional form and the generation of a first order task tree. These are described below, with the help of the example first order problem shown in Figure 3.27. The initial plan and a single method are shown; methods for the *drive*, *pickup* and *dropoff* tasks are ignored.

$$\begin{aligned}
 &\langle \text{plan}, \text{null}, \{ \langle \text{task}_1, \text{deliver}(a, \text{loc}_2), \{\} \rangle, \langle \text{task}_2, \text{deliver}(b, \text{loc}_3), \{\} \rangle \}, \\
 &\quad \{\}, \{ \langle \text{post}_1, \text{at}(a, \text{loc}_1) \rangle, \langle \text{post}_2, \text{at}(b, \text{loc}_2) \rangle \}, \\
 &\quad \{ \langle \text{post}_1, \text{post}_2, \{e\} \rangle, \langle \text{post}_1, \text{task}_1, \{b, m\} \rangle, \langle \text{post}_1, \text{task}_2, \{b, m\} \rangle \}, \{\}, \{\} \rangle \\
 \\
 &\langle \text{method}_1, \text{deliver}(\text{?package}, \text{?des}), \{ \\
 &\quad \langle \text{task}_3, \text{drive}(\text{?src}, \text{?vehicle}), \{\} \rangle, \langle \text{task}_4, \text{pickup}(\text{?package}), \{\} \rangle, \\
 &\quad \langle \text{task}_5, \text{drive}(\text{?des}, \text{?vehicle}), \{\} \rangle, \langle \text{task}_6, \text{dropoff}(\text{?package}), \{\} \rangle \}, \{\}, \{\}, \\
 &\quad \{ \langle \text{task}_3, \text{task}_4, \{m\} \rangle, \langle \text{task}_4, \text{task}_5, \{m\} \rangle, \langle \text{task}_5, \text{task}_6, \{m\} \rangle \}, \\
 &\quad \{ \text{?package} : \{a, b, c\}, \text{?des} : \{ \text{loc}_1, \text{loc}_2, \text{loc}_3 \}, \\
 &\quad \quad \text{?vehicle} : \{ \text{car}, \text{van}, \text{train} \}, \text{?src} : \{ \text{loc}_1, \text{loc}_2, \text{loc}_3 \} \}, \\
 &\quad \{ \langle \text{?src}, \text{?des}, \text{ne} \rangle \} \rangle
 \end{aligned}$$

Figure 3.27: Initial plan and example method from a first order package delivery domain. Tasks are shown in orange; state constraints are shown in blue; planning variables are shown in green; temporal and binding constraints are shown in black.

In the example problem, the agent must create a plan to deliver two packages. The initial positions of the packages are shown as blue postconditions, while their goal destinations are encoded as orange *deliver* tasks. Package *a* starts at *loc*₁ and needs to be delivered to *loc*₂, and package *b* starts at *loc*₂ and needs to be delivered to *loc*₃. The implementation of *deliver* involves a sequence of four subtasks and, importantly, four planning variables. Two variables, *?package* and *?des*, are bound to objects in the *action* of the method during task tree generation, while two free variables, *?vehicle* and *?src*, can be bound by the agent during planning.

Compilation to propositional form In this approach, the problem is compiled into propositional form during task tree generation and pMPF is used as a planning mechanism. The generation algorithm is similar to that in Figure 3.12. When a method is added to the tree, the variables in its *action* field are bound to match the arguments of the relevant task, and a propositional decomposition is produced for every remaining combination of groundings of its free variables.

Consider, for example, $method_1$ applied to $task_1$ above. A decomposition would be produced for each of the following combinations of groundings:

$$\begin{array}{llll} ?package = a & ?des = loc_2 & ?vehicle = car & ?src = loc_1 \\ ?package = a & ?des = loc_2 & ?vehicle = car & ?src = loc_3 \\ ?package = a & ?des = loc_2 & ?vehicle = van & ?src = loc_1 \\ ?package = a & ?des = loc_2 & ?vehicle = van & ?src = loc_3 \\ ?package = a & ?des = loc_2 & ?vehicle = train & ?src = loc_1 \\ ?package = a & ?des = loc_2 & ?vehicle = train & ?src = loc_3 \end{array}$$

The initial conditions of the problem (blue postconditions in the initial *plan*) state that package a starts at loc_2 , intuitively making half of these decompositions pointless. However, these decompositions are not *inconsistent* and may be necessary if a package needs to be moved more than once, so they form a valid part of the initial task tree. Similarly, the following decompositions are added for $task_2$:

$$\begin{array}{llll} ?package = b & ?des = loc_3 & ?vehicle = car & ?src = loc_1 \\ ?package = b & ?des = loc_3 & ?vehicle = car & ?src = loc_2 \\ ?package = b & ?des = loc_3 & ?vehicle = van & ?src = loc_1 \\ ?package = b & ?des = loc_3 & ?vehicle = van & ?src = loc_2 \\ ?package = b & ?des = loc_3 & ?vehicle = train & ?src = loc_1 \\ ?package = b & ?des = loc_3 & ?vehicle = train & ?src = loc_2 \end{array}$$

The number of nodes in a compiled tree is $O((dfv)^h)$ where d is the average number of decompositions per task, f is the average number of free variables per method, v is the average number of values per free variable and h is the height of the tree. First order planning domains with nested method calls, particularly recursive domains (Section 3.6), can give rise to extremely large compiled task trees.

Expansion in first order form An alternative approach is to create a first order task tree in which planning variables are preserved. Decompositions are produced by cloning methods and uniquely renaming the variables therein. This approach produces significantly smaller task trees than compilation to propositional form, as only one decomposition is added to each task for each applicable method.

Consider, again, $method_1$ in the example problem above. When it is used to expand $task_1$, a decomposition $decomp'$ might be produced with variables as follows:

$$\begin{array}{ll} ?package \rightarrow ?package' & ?des \rightarrow ?des' \\ ?vehicle \rightarrow ?vehicle' & ?src \rightarrow ?src' \end{array}$$

When the decomposition is added to $decomps(task_1)$, *frame binding constraints* are added to $bindings(decomp')$ to make sure $action(task_1)$ has the same set of groundings as $action(decomp')$:

$$bindings(decomp') \Leftarrow \{ \langle a, ?package', eq \rangle, \langle loc_3, ?des', eq \rangle \}$$

When $method_1$ is used to expand $task_2$, a decomposition $decomp''$ might be produced. This decomposition would have a different set of variables:

$$\begin{array}{ll} ?package \rightarrow ?package'' & ?des \rightarrow ?des'' \\ ?vehicle \rightarrow ?vehicle'' & ?src \rightarrow ?src'' \end{array}$$

with equivalent frame binding constraints in $bindings(decomp'')$:

$$bindings(decomp'') \Leftarrow \{ \langle b, ?package'', eq \rangle, \langle loc_3, ?des'', eq \rangle \}$$

The number of nodes in a first order task tree is $O(d^h)$ where d is the average number of decompositions per task and h is the height of the tree. Assuming compilation and expansion produce task trees of the same height, a compiled propositional tree will always be a factor of $O((fv)^h)$ larger than its first order equivalent. This assumption is valid for all non-recursive and recursive problems (Section 3.6) as they are handled in this thesis.

Summary conditions First order task trees still produce propositional summary information. Propositional summary conditions can be merged as they are propagated up the task tree (Section 3.3.9). This is useful as it removes unnecessary detail, preserving only the information necessary for the analysis of summary relationships at high levels of abstraction. Planning variables are defined in task networks throughout the task tree: if first order summary conditions were produced, the variables in them would lose meaning as they were propagated towards the root of the tree, preventing merging and resulting in a large amount of meaningless information.

The first order task tree approach is not without its disadvantages. Changes to the relationships between widely used planning variables can require large amounts of summary information to be recalculated. This slows down planning and increases the computational cost of summary based heuristics. However, despite this disadvantage, there are many “traditional planning problems” that cannot be represented in propositional form using task trees in a feasible amount of memory. An empirical comparison of performance using the compilation and first order approaches is presented in Section 5.3. While first order task trees alleviate the problem, memory overheads are still a major disadvantage of task trees compared to “traditional” HTN mechanisms.

3.5.3 Planning with first order task trees

Despite the increased complexity of first order task trees, many of the features of the propositional planning mechanism can be reused with minor modification.

Modified consistency test The `is_consistent` function from Section 3.4.1 needs to be amended to detect planning variables that have no possible values due to over-constrained binding constraints. The first order definition is shown below, with new terms highlighted in blue:

$$\begin{aligned} \text{is_consistent}(net) = & \forall node, node' \in \text{children}(net) \text{ tlookup}(node, node', net) \neq \emptyset \\ & \wedge \forall var \in \text{vars}(net) \text{ values}(var, net) \neq \emptyset \\ & \wedge \forall task \in \text{tasks}(net) \ |decomps(task)| \geq 1 \\ & \wedge \forall decomp \in \text{decomps}(task \in \text{tasks}(net)) \ \text{is_consistent}(decomp) \end{aligned}$$

Planning operator for binding variables When a task is decomposed, extra binding constraints are added to the root of the task tree to constrain bound variables from the decomposition appropriately. A new planning operator, `bind_op`, is introduced to make two actions or literals equal or not equal. The operator is shown in Figure 3.28. Note that if the variables being bound are used widely in the task tree, the *dirty* bit may be set in a large number of task networks and a significant amount of summary information may need to be recomputed (see below).

```

1  function bind_op(action1, action2, rel, net):
2      for each arg1 ∈ args(action1), arg2 ∈ args(action2):
3          bindings(net) ← {⟨arg1, arg2, rel⟩}
4      for each changed var ∈ vars(net):
5          propagate changes of values to equivalent variables in descendants of net
6          set dirty bit in each member of descendants and all their ancestors

```

Figure 3.28: Algorithm for the first order bind_op planning operator. Despite their names, *action*₁ and *action*₂ can be actions or literals. *rel* is one of {*eq*, *ne*}.

Modified decompose operator The decompose_op planning operator needs to be modified so that when a task is decomposed, the planning variables and binding constraints from the decomposition are merged with those in the task’s parent task network. The first order version of the operator is shown in Figure 3.29.

```

1  function decompose_op(net, task):
2      delete task from tasks(net)
3      for each node ∈ children(decomp ∈ decomps(task)):
4          add child to children(net)
5          temporal(net) ← {⟨id(task), id(child), {e, si, fi, di}⟩}
6      for each arg ∈ vars(decomp ∈ decomps(task)):
7          add var to vars(net)
8      for each con ∈ bindings(decomp ∈ decomps(task)):
9          bindings(net) ← {con}
10     bind_op(action(task), action(decomp))
11     for each con ∈ temporal(decomp ∈ decomps(task)):
12         temporal(net) ← {con}

```

Figure 3.29: Algorithm for the first order decompose_op planning operator. “Extra” lines not present in the propositional version are shown in blue.

Modified decompose refinement The decompose_ref refinement needs to be modified so that it grounds the action of a task in addition to selecting a particular decomposition. This makes the decompose_ref refinement very similar to the propositional decompose_ref refinement applied to a compiled propositional task tree. The first order version of this refinement is shown in Figure 3.30.

With these features of fMPF, agents are able to solve first order non-recursive HTN planning problems using task trees and summary information. The next section introduces techniques for

```

1  function decompose_ref(plan, task) → newplans:
2      for each decomp ∈ decomps(task):
3          for each grounding g shared between action(task) and action(decomp):
4              create a copy newplan of plan
5              locate equivalent newtask and newdecomp for task and decomp
6              select_op(newtask, newdecomp)
7              bind_op(action(newtask), g, eq, newplan)
8              add newplan to newplans

```

Figure 3.30: Algorithm for the first order decompose_ref refinement. “Extra” lines not present in the propositional version are shown in blue.

handling a restricted set of recursive first order problems, either with fMPF or after compilation to pMPF.

Overheads of planning with first order trees Every time `bind_op` is used, the domains of one or more planing variables change in the root node of the tree. Summary information has to be recomputed for any node that references one of the changed variables. This in turn makes further recomputation necessary as the new summary information is propagated up the tree.

The amount of summary information is limited by two factors in practice. Firstly, `bind_op` is only used as part of `decompose_ref` (ordering refinements are left unaffected). Secondly, summary information only has to be recomputed when the domains of variables change: typically, only a small fraction of the variables in the tree are affected each decomposition, and these will be referred to in a fraction of the branches of the tree. However, summary recalculation still imposes an overhead that is not present in propositional planning.

In the worst case, assuming that *all* of the summary information in the tree has to be recalculated *every* refinement, the planner suffers from an $O(d^{(h+3)}c^3)$ operation overhead per iteration. Much of the efficiency of first order planning depends on the extent to which this overhead can be minimised through careful implementation and optimisation.

3.6 Recursive problems

Many HTN planning problems involve recursive actions. For example, in Blocksworld the clearing of a block is recursive:

In order to pick up block A I have to make sure there is nothing on top of it.

If there is a block B on top of A I must pick it up and place it on the table.

In order to pick up block B I have to make sure there is nothing on top of it.

If there is a block C on top of B I must pick it up and place it on the table...

as is travelling from location to location in navigation and logistics domains:

To get from A to Z I first go to B and then get from B to Z.

To get from B to Z I first go to C and then get from C to Z...

and moving a stack of disks¹⁵ in Towers of Hanoi:

$$\text{move}(5, A, C) \Rightarrow (\text{move}(4, A, B), \text{move}(1, A, C), \text{move}(4, B, C))$$

$$\text{move}(4, A, B) \Rightarrow (\text{move}(3, A, C), \text{move}(1, A, B), \text{move}(3, C, B))...$$

Recursive tasks complicate task tree generation because it is difficult to put a bound on the size of the tree before planning starts. However, while recursion may suggest an infinite task tree, for first order problems with a fixed finite number of world objects this is never the case. For example:

- in a simple n block Blocksworld problem, a maximum $n - 1$ blocks need to be moved to the table to clear a particular block;
- in an n location navigation problem, a maximum $n - 2$ locations need to be visited *en route* from A to B;
- in an n disk Towers of Hanoi problem, only n levels of recursion are necessary to move all the disks from one pillar to another.

¹⁵ $\text{move}(x, y, z)$ denotes the abstract task of moving x disks from pillar y to pillar z .

For any problem there is an optimal task tree size that suits the problem. If too small a tree is generated it may not contain any solution plans. If the tree is too large, agents waste time during planning maintaining information that is unnecessary.

As well as being necessary for summarisation, a fixed maximum tree size can be useful for implementing some search algorithms, including search algorithms based on constraint satisfaction techniques (Section 4.4.1). This section discusses a task tree generation technique called *value counting* that allows the estimation of a suitable initial size for first order and compiled propositional task trees in certain recursive problems.

3.6.1 Estimating optimal tree size

It is difficult to calculate the optimal initial size of a task tree without reverse engineering a valid solution to the problem. However, it is sometimes possible to make a good guess without problem specific knowledge, using a novel technique called *value counting*.

$$\begin{aligned} &\langle travel_1, travel(?src, ?des), \{\}, \{\}, \\ &\quad \{\langle task_1, move(?src, ?aux) \dots \rangle, \langle task_2, travel(?aux, ?des) \dots \rangle\}, \dots \rangle \\ &\langle travel_2, travel(?src, ?des), \{\langle pre_1, edge(?src, ?des) \rangle\}, \{\}, \langle move(?src, ?des) \dots \rangle, \dots \rangle \\ &\langle move_1, move(?src, ?des), \{\langle pre_1, edge(?src, ?des) \rangle, \langle pre_2, at(?src) \rangle\}, \\ &\quad \{\langle post_1, \neg at(?src) \rangle, \langle post_2, at(?des) \rangle\}, \{\}, \dots \rangle \end{aligned}$$

Figure 3.31: Abbreviated methods for single robot navigation. Task networks and temporal constraints are written in black, tasks in orange, and state constraints in blue.

Consider, for example, the methods for single robot navigation problem shown in Figure 3.31. The abstract action *travel* represents movement between distant locations, possibly via a number of intermediate locations, while the primitive action *move* represents movement between adjacent locations along a connecting edge.

travel tasks can be decomposed using the recursive method $travel_1$ or the non-recursive method $travel_2$. Binding constraints make sure that in $travel_1$, $?src \neq ?aux$, and in $travel_2$, $?aux \neq ?des$ ¹⁶. Figure 3.32 intuitively shows that for any configuration of connections on an n location graph, the maximum number of primitive *moves* required to perform a single *travel* task is $n - 1$.

¹⁶The case of *null* movement where $?src = ?des$ is ignored.

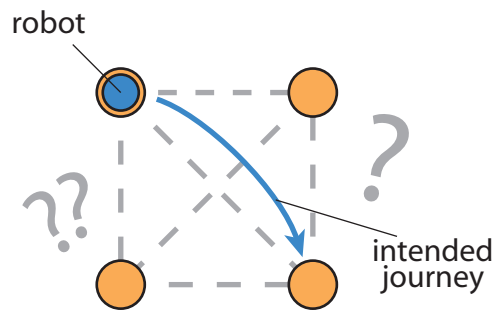


Figure 3.32: Single robot navigation problem. Each journey on a graph containing n locations requires a maximum $n - 1$ moves.

This information can be retrieved by counting the number of possible values of free variables in level 1 tasks in the task tree. $travel_1$ is the only recursive method in the domain, and $?aux$ is the only free variable in $travel_1$. The binding constraints in $travel_1$ dictate that, if $?src$ and $?des$ are fixed, there are $n - 2$ possible values for aux .

The value counting approach builds a task tree containing $n - 2$ levels of recursion of $travel_1$. The resulting tree is shown in Figure 3.33. The tree represents a disjunction of routes of length 1 to $n - 1$, through any set of locations in any order.

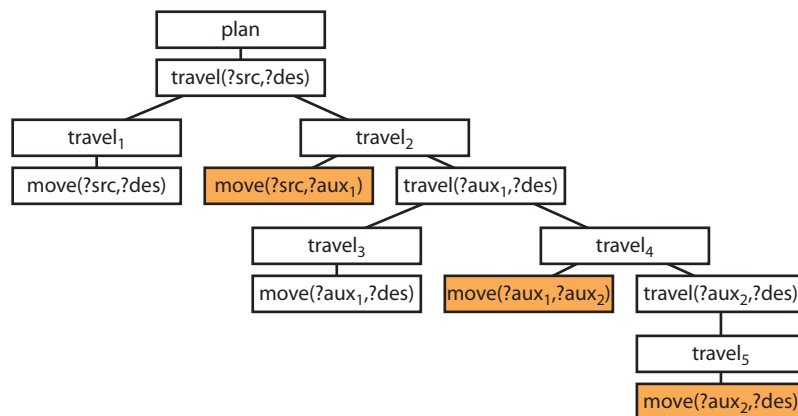


Figure 3.33: Initial task tree for the problem in Figure 3.32. Primitive tasks used in the worst case of $n - 1$ moves are highlighted in orange.

Action independence Value counting only works in problems with a certain structure. To ensure value counting will work in all cases, the (human) domain programmer needs to structure

recursive tasks and methods such that *the subtree of any level 1 task contains solution histories for any set of initial conditions*. This will hereafter be referred to as the *action independence assumption*. Examples are given below.

Action independence is specified for each abstract action in the problem domain. In Figure 3.31 the *travel* action is independent because the *travel₁* and *travel₂* methods guarantee that, for every solvable problem, a level 1 *travel* subtree will contain a history that constitutes a valid solution plan. The problem may be unsolvable with value counting if any level 1 tasks do not satisfy the action independence assumption.

Action independence does not guarantee that a plan for a given abstract action can be found: it simply guarantees the maximum recursion depth whenever a plan does exist. For example, in Figure 3.32 there may be *no* edges connecting the locations on the graph, making journeys between different locations impossible. If a plan *can* be found, however, it is guaranteed to be contained within the generated task tree.

```

⟨travel1, travel(?robo, ?src, ?des), {}, {},
  {⟨task1, move(?robo, ?src, ?aux) ...⟩, ⟨task2, travel(?robo, ?aux, ?des) ...⟩}, ...⟩
⟨travel2, travel(?robo, ?src, ?des), {⟨pre1, edge(?src, ?des)⟩}, {},
  {⟨move(?robo, ?src, ?des) ...⟩}, ...⟩
⟨move1, move(?robo, ?src, ?des),
  {⟨pre1, edge(?src, ?des)⟩, ⟨pre2, at(?robo, ?src)⟩, ⟨pre3, clear(?des)⟩},
  {⟨post1, ¬at(?robo, ?src)⟩, ⟨post2, at(?des)⟩, ⟨post3, clear(?src)⟩, ⟨post4, ¬clear(?des)⟩},
  {}, ...⟩

```

Figure 3.34: Abbreviated methods for multi robot navigation. Differences from the single agent methods in Figure 3.31 are shown in **green**. The *clear* pre-/postconditions of *move₁* prevent two robots being co-located or passing on the same edge.

Action independence is a restrictive assumption and building compliant domains is not always possible. Consider the set of methods for multi robot navigation shown in Figure 3.34. The *clear* pre-/postconditions of the *move₁* method make it impossible for two robots to be co-located or pass on the same edge. *travel* is no longer an independent action with these state constraints, as the *clear* precondition is not explicitly achieved as part of any of the methods in *travel* subtrees. It is no longer certain whether or not a given level 1 *travel* action will be achievable within the estimated number of *moves*.

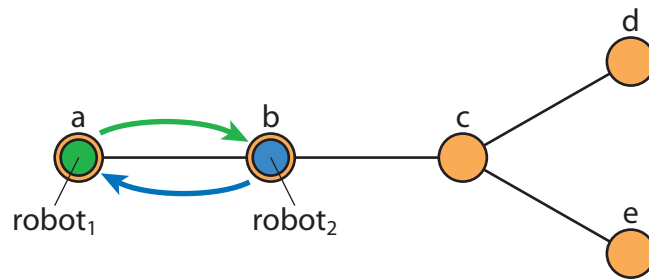


Figure 3.35: Multi robot navigation problem demonstrating shortcomings of value counting.

Consider the example in Figure 3.35. In this example, $robot_1$ and $robot_2$ must exchange places. They cannot pass each other directly because of the *clear* pre/postconditions of the $move_1$ method. To solve the problem, the robots must use locations d and e as passing places. Ignoring inter-robot temporal orderings, there are two solutions to the problem, shown in Figure 3.36.

Solution 1: $robot_1$ $a \rightarrow b \rightarrow c \rightarrow d \rightarrow c \rightarrow b$
 $robot_2$ $b \rightarrow c \rightarrow e \rightarrow c \rightarrow b \rightarrow a$
 Solution 2: $robot_1$ $a \rightarrow b \rightarrow c \rightarrow e \rightarrow c \rightarrow b$
 $robot_2$ $b \rightarrow c \rightarrow d \rightarrow c \rightarrow b \rightarrow a$

Figure 3.36: Solutions to the problem in Figure 3.35.

In both of these solutions, the required number of *moves* for each robot is 5, which is one more than the 4 predicted by value counting. The situation can be made worse by adding locations between b and c on the graph; the number of *moves* required per robot increases by 2 for every location added, whereas the prediction from value counting only increases by 1.

This problem can be avoided by changing the method structure from Figure 3.34 so that it obeys action independence. This can be done, for example, by adding the abstract action $travel_avoid(?robo, ?src, ?des)$ shown in Figure 3.37. This action, which simply doubles the maximum number of *move* tasks required in general to move from a to b , allows either robot in a two robot problem to travel while avoiding its team mate. $travel_avoid$ is the only action that can be safely used in level 1 tasks in two robot navigation problems.

The methods in Figure 3.37 have two major drawbacks:

```

⟨avoid1, travel_avoid(?robo, ?src, ?des), {}, {},
  {⟨task1, travel(?robo, ?src, ?aux)⟩, ⟨task2, travel(?robo, ?aux, ?des)⟩}, ...⟩
⟨avoid2, travel_avoid(?robo, ?src, ?des), {}, {}, {⟨task1, travel(?robo, ?src, ?des)⟩}, ...⟩

```

Figure 3.37: Methods for the independent two robot navigation action, *travel_avoid*.

1. The methods only work in two robot problems. Versions for three or more robots could be created by adding tasks and *?aux* destinations to the *avoid₁* method, but in general the specification of such methods requires a planning formalism that supports universal quantification. The shortcomings of value counting in this example stem from the fact that the *clear* precondition of *move₁* implies that the absence of a robot is required for movement to be possible, but does not provide a variable of type *robot* with which to enumerate the robots that must be avoided.
2. The methods require one agent to have control over both robots in the domain. If agents have control over different robots, subtasks of *travel_avoid* will have to be *contracted out* to the relevant team mates.

Units of recursion *Recursion interval* and *recursion factor* are two important properties of a method, both defined in terms of a subset of a task tree called a *unit of recursion*. Informally, this is the part of a task tree that repeats for any given recursive method.

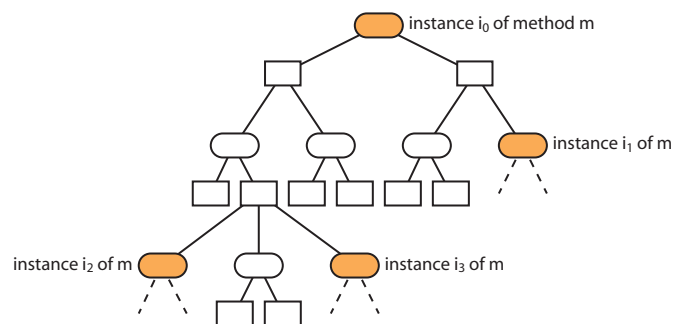


Figure 3.38: The unit of recursion of a method *m*, taken from an instance *i₀* down to the next instances *i₁*, *i₂* and *i₃*. The recursion factor and recursion interval of *m* in this example are 3 and 4 respectively.

The unit of recursion of a method *m*, shown in Figure 3.38, is the top part of the first order

subtree of an instance i_0 of m , taken down to the next instances of m in the tree¹⁷. The recursion interval of m is the maximum distance from i_0 to one of the other instances of m in the unit; the recursion factor of m is the number of instances of m in the unit other than i_0 . These measures are important because they affect respectively the height and branching factor of task trees generated containing m (Section 3.7).

3.6.2 Adaptive task tree resizing

As mentioned above, a significant class of problems that do not satisfy action independence includes problems where agents are mutually dependent on each other to achieve some task preconditions. An example of this is the “airlock” problem described in Section 4.4.2, in which robots are required to open doors for one another: it is impossible for this problem to satisfy action independence as a single task subtree cannot be created involving the movements of both robots. This is a severe limitation on the applicability of value counting to interesting multi agent problems, as many actions that allow limited control of resources *cannot* obey action independence: value counting is not guaranteed to produce initial task trees that are large enough in these situations.

A more flexible alternative to value counting, described in (Gurnell, 2004)¹⁸, involves creating an initially small task tree and *expanding* it as necessary during planning to resolve conflicts. This means a maximum bound cannot be placed on task tree size, and has difficult consequences for heuristic values (including `can_any_way` and `might_some_way`) as some summary information may initially be missing from the task tree. This approach is the subject of ongoing research.

3.7 Task tree generation algorithm

The value counting algorithm for task tree generation is shown in Figure 3.39. The `expand` function takes three parameters: a *task* to expand, the set of *methods* in the library of *owner(task)*, and a Boolean specifying whether or not to make a *propositional* tree. If *propositional* is *true*,

¹⁷This tree fragment will be identical for any instance of m in the tree.

¹⁸Available from <http://www.cs.bham.ac.uk/~djg>.

the function produces a compiled propositional tree (Section 3.5.2). Otherwise, a first order tree is produced. The function `groundings` takes a task network as a single parameter and returns a copy for each consistent grounding of its free planning variables. The `depth(task, method)` function returns the *recursion depth* of the relevant task, which is defined as 1 plus the number of ancestor task networks of *task* that were derived from *method*.

```

1  function expand(task, methods, propositional):
2      for each method ∈ methods:
3          if unify(action(task), action(method)) ≠ null:
4              if num_groundings(method) > depth(task, method):
5                  decomp ← copy(method)
6                  for each (arg1, arg2) ∈ (action(task), action(decomp))
7                      if arg1 is a variable:
8                          add arg1 to vars(decomp)
9                          vars(decomp) ← {⟨arg1, arg2, eq⟩}
10                 if propositional:
11                     decomp(task) ← decomp(task) ∪ groundings(decomp)
12                 else:
13                     decomp(task) ← decomp(task) ∪ decomp
14                 for each grandchild ∈ tasks(decomp ∈ decomp(task)):
15                     expand(grandchild, methods, propositional)

```

Figure 3.39: Algorithm for first order and compiled propositional task tree generation.

The `expand` function is called for each level 1 task of an initial joint plan to create a joint task tree. Once the task tree has been built it must be simplified to remove inconsistent task networks and decompose single decomposition abstract tasks (Section 3.4.4).

As shown in Section 3.6.1, the number of nodes in compiled and expanded task trees is $O((dfv)^h)$ and $O(d^h)$ respectively, where d is the average number of decompositions per task, f is the average number of free variables per decomposition, v is the average number of values per free variable, and h is the height of the tree. The height of trees generated by value counting is $O(iffv)$ where i is the maximum recursion interval of any method in the problem. First order task trees produced by value counting are thus smaller than equivalent compiled trees by a factor of $O((fv)^{iffv})$.

3.8 Summary of planning mechanisms

The discussion of planning mechanisms is now complete. To summarise, a common mechanism called MPF was described on which various multi agent planning algorithms can be implemented and compared. Requirements for the mechanism were outlined in Section 3.1, and the *Concurrent Hierarchical Plans (CHiPs)* mechanism of Clement (2002) was suggested as a starting point in Section 3.2. The advantages and limitations of CHiPs were outlined in Section 3.2.5. CHiPs has many advantages, but also some disadvantages that were addressed in MPF:

- it has no mechanism for representing ownership of tasks and constraints by different agents;
- it is only capable of representing non-recursive propositional planning problems.

Sections 3.3 and 3.4 discussed a propositional planning formalism called pMPF that is based closely on CHiPs. pMPF has two novel features of note:

- a simple mechanism is implemented for representing ownership of tasks and constraints, allowing joint plans to be split into sets of coordinated individual plans and vice versa;
- interval temporal algebra is used, allowing a more flexible representation of partial action orderings that reduces the branching factor during planning.

Section 3.5 extended pMPF to produce a new planning mechanism, fMPF, that is capable of representing first order planning problems. Section 3.6 introduced a way of handling a restricted set of recursive planning problems in fMPF, using a task tree generation technique called *value counting*.

fMPF is used as the planning mechanism for the rest of this thesis. A brief empirical comparison of tree generation and planning with pMPF and fMPF is made in Section 5.3.

Chapter 4

Planning algorithms

This chapter presents algorithms for implementations of the centralised planning, plan merging and distributed local planning approaches from Chapter 2, based on the MPF mechanisms described in Chapter 3. The centralised planning algorithm is adapted from the centralised plan coordination algorithm of Clement (2002), the plan merging algorithm is an adaptation of the centralised planning algorithm, and the distributed local planning algorithm is a novel blend of heuristic depth first search and distributed constraint satisfaction that is inspired by the work of Yokoo and Hirayama (2000). The approaches can be fairly compared because of their use of the common mechanisms: an empirical comparison is conducted in Section 5.4.

Similarities between algorithms The algorithms presented in this chapter can be used with pMPF and fMPF planning mechanisms alike by substituting in the appropriate definitions of refinements and planning operators from Chapter 3. While the algorithms are all based on the same planning mechanism, they use different search algorithms and different techniques for coordination and recovery from failure.

The notation from Chapter 3 is used again to describe data structures and algorithms throughout this chapter. A guide to the notation can be found in Appendix A.

4.1 Client-server model

The definition of an agent quoted at the beginning of Chapter 1 states that an agent is “*a system situated [in] an environment that senses the environment and acts on it [...] in pursuit of its own agenda*”. This implies that an agent is a self contained reasoning system with its own senses, goals and internal data stores. To preserve this notion, the planning algorithms introduced in this chapter are implemented as *client / server systems*. Each agent is run on a separate computer, ensuring that it is as independent of the rest of the system as possible, and that agents are given the same amount of processing power independent of the choice of approach and size of the team. Multi CPU implementations such as this give teams of agents more processing power than single CPU implementations, but incur an extra cost in inter-agent communication. This thesis does not investigate the effect of different implementations on algorithmic performance, although it would be a useful addition to the work presented.

Role of the server The server computer initiates the planning process, parses the problem description file and generates an initial joint task tree. It then waits for the required number of client computers to connect (one client per agent) and provides suitable initial individual task trees for them to download. Once planning has started, the server acts as a communication channel for the clients until a solution has been found or one or more clients has signalled failure. It then collects solution plans if they are available and outputs a final joint plan and any relevant experimental data.

While the plan server is capable of combining individual plans to produce a joint plan, it does not remove conflicts between plans. *Plan merging*, the process of taking a set of uncoordinated plans and removing conflicts to produce a coordinated joint plan, is done by a plan merging agent, running on a client computer.

Role of the clients Each client runs on a separate computer on the network and supports a single planning or merging agent. Centralised planning problems require a single client computer, distributed problems require one client for each of the n agents in the problem, and plan-then-merge problems require $n + 1$ clients (n clients for planning plus 1 for plan merging). Where applicable, clients communicate with each other through the exchange of algorithm specific

messages and summary information. Messages are broadcast or sent point-to-point, using the server as an intermediary equivalent to a central email server. In the current implementation all clients must be registered before planning is allowed to begin, although this is not a strict requirement of any of the algorithms presented.

4.2 Centralised planning

The simplest way of solving a multi agent planning problem is to pool all knowledge of goals, state constraints, tasks and methods and pass everything to a single *delegate planning agent* (Section 2.2.1). The delegate uses all this knowledge to create a single joint plan for the whole team, and then redistributes individual plans back to the individual agents.

The centralised planning algorithm described below is similar to the *centralised coordination algorithm* described for CHiPs (Chapter 6 of Clement, 2002). In both algorithms, the process of merging starting information and distributing final individual plans is ignored: the plan server simply creates an initial task tree and a single client agent uses it to find a joint plan.

Strategy for refinement choice Clement defines a strategy for refinement choice called *Expand Most Threats First (EMTF)*, in which the task with the most threats is always selected first for decomposition¹. This ensures that the agent is always attempting to resolve flaws involving the task with the most predicted threats. The centralised planning approach here uses the same technique for the same reason. The threats on each tasks are counted and an appropriate refinement is applied to the task with the highest count. If the task is abstract, lifted or has more than one decomposition, the `decompose_ref` refinement is used. Otherwise, the `order_ref` refinement is used.

Communication There is no communication in this algorithm apart from the download of the initial task tree from the server and the final upload of the solution plan.

¹As mentioned in a footnote in Section 3.1, the verb “to decompose” refers to the same thing as the verb “to expand” used by Clement. In the interests of consistency, the name of the EMTF heuristic has been kept the same despite this terminological difference.

Planning algorithm The algorithm itself is a simple implementation of Kambhampati’s refinement planning algorithm from Section 2.1.1. The `is_solution` function is implemented with the `can_any_way` function from Section 3.4.1, and the `get_refinement` function is implemented with the EMTF strategy above. The `might_some_way` function from Section 3.4.1 is used when applying planning operators to immediately discard inconsistent plans and plans with unresolvable threats.

```

1  function find_plan(initial) → solution:
2      create empty open_list
3      open_list ← push(initial, open_list)
4      while ¬ empty(open_list):
5          (plan, open_list) ← pop(open_list)
6          if can_any_way(plan):
7              solution ← plan
8              return
9          else if might_some_way(plan):
10             ref ← pick_refinement(plan)
11             open_list ← push(plans(ref), open_list)
12     solution ← failure

```

Figure 4.1: Algorithm for centralised planning.

The complete centralised planning algorithm is shown in Figure 4.1. Best first (BFS) and heuristic depth first (DFS) versions of the algorithm can be implemented using different implementations of the push and pop operators:

DFS uses stack like definitions: push adds to the front of the open list and pop removes from the front (last in first out).

BFS uses priority queue like definitions²: push adds plans such that the list remains sorted according to ascending heuristic value and pop removes from the beginning of the list where the value is lowest. In MPF the heuristic value used to sort the open list is the number of threats to level 1 summary conditions, so the planner always considers the open plan with the fewest threats first.

The centralised planning and “plan-then-merge” algorithms described in this chapter are both

²Note that BFS stands for Best First Search: breadth first search is not considered here.

based on the BFS variant of this algorithm. The distributed local planning is based on the DFS variant, for reasons that will be discussed in Section 4.4.

Agent independence This approach is the least sensitive of the three to the privacy and independence of the planning agents. Agents have to submit complete information about their initial task trees to the delegate agent before they start planning. This gives the delegate agent complete access to the knowledge, goals and planning capabilities of the entire team. Agents also effectively hand over all of their ability to make choices during planning, removing any independence. Some independence may be regained by making the delegate produce several candidate solutions and allowing the agents to negotiate over the final choice, although this approach is not investigated in this thesis.

4.3 Plan-then-merge

An alternative to centralised planning is for each agent to create a plan in isolation, ignoring the goals and plans of the rest of the team. The agents submit their final individual plans to a *delegate plan merging agent*, which uses a variant of the planning algorithm above to coordinate them. The coordinated individual plans are returned to the individual agents ready for execution.

Plan merging is a common approach to multi agent planning. The implementation described here involves two stages, *planning* and *merging*, although in principle approaches with multiple planning and merging stages are also possible. The plan server distributes initial individual task trees to n planning agents, which use them to create uncoordinated individual plans. These plans are then uploaded and merged into an initial joint task tree, which is sent to the merger agent. The merger resolves conflicts in this tree and uploads a final coordinated joint plan.

Planning Planning is performed individually by the agents in the problem. Each agent is aware only of the initial conditions of the problem and the tasks in its own plan. Agents download initial individual task trees and use the centralised planning algorithm to create a solution plan. When finished, they upload their solutions to the server.

Plan merging Because the agents in the planning phase create plans without regard for each others' goals, knowledge, abilities and so on, there is a high chance that their individual solutions will contain redundant or conflicting tasks. The *merging* stage involves taking the set of individual plans and removing conflicts to create a coordinated joint plan.

Plans are merged using a variation of the centralised planning algorithm. The server creates an *initial joint task tree* from the individual solutions uploaded by the planning agents. A *merger* agent downloads the tree and uses the centralised planning algorithm to create a solution plan to the complete joint planning problem. The final joint solution is then uploaded to the server.

Communication Apart from the downloading of initial task trees and the uploading of final plans, there is no communication in this model. The planning agents, in particular, are oblivious to each others' presence.

An example Consider the following example from the Blocksworld. Three agents are involved in a plan-then-merge problem. Two of these agents, *planner₁* and *planner₂*, are planning agents. The third, *merger*, is a plan merging agent³.

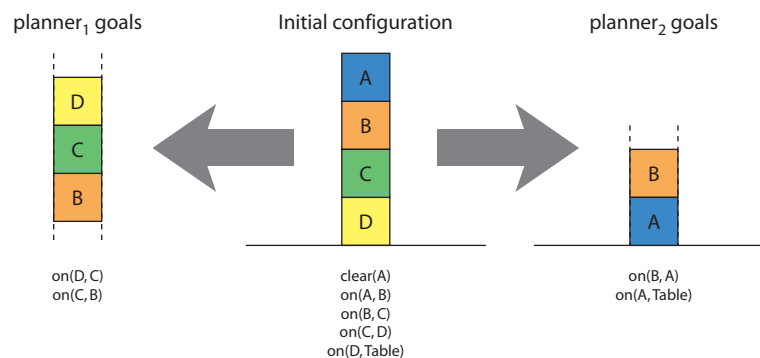


Figure 4.2: Two agent Blocksworld problem suitable for the plan-then-merge approach.

The initial conditions of the problem and the goals of *planner₁* and *planner₂* are shown in Figure 4.2. Assuming that both individual goal states have to be satisfied at the same time, the joint problem is equivalent to a reversal of the initial tower. Given this input, the agents may be expected to produce plans similar to those in Figure 4.3.

³In an actual application the role of *merger* may be taken on by one of the planning agents.

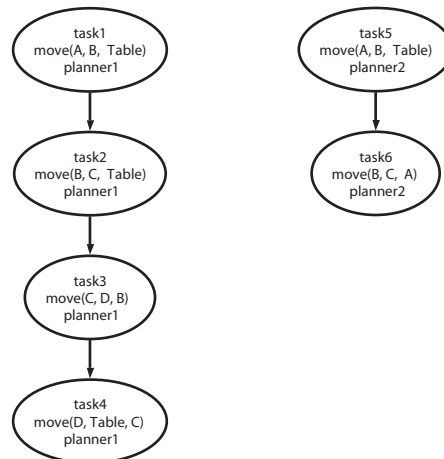


Figure 4.3: Solutions to the individual problems in Figure 4.2. *move* actions are in the format *move(?block, ?source, ?destination)*. Arrows denote temporal orderings.

These plans are conflicting and cannot simply be unioned to form a joint plan. There are two conflicts:

- $task_1$ and $task_5$ contain the same redundant action $move(A, B, Table)$;
- $task_2$ and $task_6$ conflicting because they move B to different locations.

Tasks must be deleted and ordering constraints imposed to coordinate and merge the plans. There are several ways of doing this, one of which is shown in Figure 4.4. In this solution $task_2$ and $task_5$ are deleted and ordering constraints are imposed on the remaining tasks to remove conflicts.

Merging task trees By choosing a suitable initial task tree, many individual plans can be merged with the centralised planning algorithm discussed in Section 4.2.

HTN planning algorithms are capable of adding ordering constraints to the joint plan to resolve conflicts between actions, but they are not strictly capable of deleting tasks from a plan. Task removal is implemented by adding a decomposition to each primitive task. The extra decomposition represents a noop that can be performed instead of the actual task. An appropriate decomposition is chosen for each task during plan merging, effectively deciding whether to keep or delete the task. Tasks with noop decompositions are deleted from the final task tree as a post-processing step.

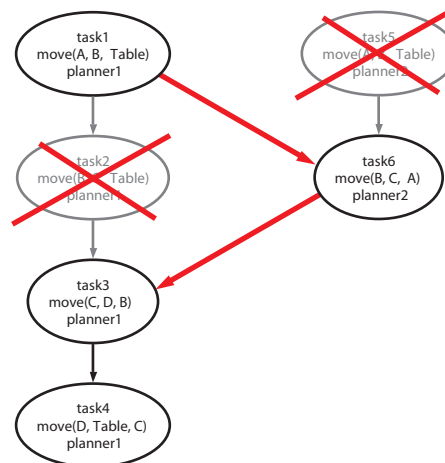


Figure 4.4: Coordinated joint plan created by merging the individual plans in Figure 4.3. Red crosses denote deleted actions and red arrows denote new ordering constraints.

This approach requires that the goals of the individual problems be stated as state constraints as well as abstract tasks. Without such a secondary specification, the merger agent would be able to create a complete conflict free joint plan simply by deleting all the tasks from the joint plan. These state constraints must be specified as part of the initial planning problem and ordered appropriately with respect to the tasks for each agent.

Consider, for example, the plans in Figure 4.3. Precondition state constraints are needed for the literals $on(D, C)$, $on(C, B)$, $on(B, A)$ and $on(A, Table)$ to make sure the desired configuration of blocks is achieved once the plans have been merged.

Efficiency of plan-then-merge By dividing a large joint problem up into a number of smaller individual problems, plan-then-merge can sometimes achieve a speed increase over centralised planning. The time taken to create individual plans can be significantly reduced because the joint problem is broken down into smaller components. A speed gain is achieved overall if the reduction in individual planning time is more significant than the time taken to merge the individual plans into a joint plan.

As discussed in Section 2.2.2, plan-then-merge will not always be a more efficient strategy than centralised planning. If individual plans have many redundant and conflicting actions, plan

merging may be prohibitively expensive. The nature of the planning and merging approaches used will also have an impact on the relative efficiency of the two techniques. Examples of this are seen in the experimental results in Section 5.4.

Agent independence Agents are completely independent during the planning phase, but this changes drastically during merging. From a privacy point of view, the agents have to share their final plans with the delegate agent before they are executed, although this is not as much of an issue as the complete knowledge sharing involved in centralised planning as goals and some knowledge can remain private. From an independence point of view, the agents are able to create individual plans according to their own specifications, but there is no guarantee they will be merged fairly. It would be possible, for example, for the delegate to favour a single agent, removing redundant actions from its plan and assigning them to other agents. “Unfair” plan merging does not have to be deliberate: the merger agent may, for example, have a different interpretation of plan quality from the planning agents.

Disadvantages of plan-then-merge The effectiveness of plan-then-merge is strongly dependent on two factors: the ability of planning agents to produce individual plans without communication and the ability of the merging agent to produce a joint plan from the set of individual plans.

Planning agents can only produce individual plans in isolation if they are not reliant on other agents to achieve preconditions. Consider, for example, the robots in the airlock problem from Section 1.2.2. If the robots do not communicate during planning they will not realise that they are able to open the airlock doors for one another.

Even if planning agents are able to find individual plans that achieve their goals, the discovery of a joint solution is dependent on the ability of the merger agent to create a joint plan. The algorithm for plan merging described above, for example, is limited because it cannot *add* tasks to or *change* tasks in the joint plan. If the planning agents produce tasks with unresolvable conflicts, the merger agent will be unable to find a solution even if there is an alternative set of non-conflicting tasks that could be chosen. This is a restriction of this specific implementation, caused by strict adherence to the HTN principle of planning-by-decomposition.

If a problem cannot be solved by plan-then-merge because of conflicts that arise during merging, it is said to be *non-mergeable*⁴ (Section 2.2.2). Non-mergeable problems can sometimes be solved using centralised planning as a fall-back, or by using a multi staged approach in which sets of compatible goals are identified and distributed to avoid merging issues. Other workarounds for this problem are discussed in Section 2.2.2.

The fall-back technique mentioned above is used in the *Plan Merging Paradigm* (Alami et al., 1997), described in Section 2.2.2, in which agents continuously generate new goals, create plans to achieve them, and then attempt to merge them with existing plans from other agents in the environment. If a plan cannot be merged with the existing plans of the team, a *deadlock* situation occurs and a centralised planner is called in to recreate the current joint plan from scratch. This is potentially a time consuming approach that is also bad from an independence point of view.

4.4 Distributed local planning

Even though some of the problems with plan-then-merge can be overcome by making changes to the merging mechanism (see above), the approach is still capable of solving strictly fewer problems than centralised planning. This is because agents do not exchange information before or during planning.

Distributed local planning attempts to overcome this limitation without compromising independence by allowing agents to exchange information about resource requirements during planning. The plan server distributes initial individual task trees to the agents, which plan and exchange messages with one another and upload coordinated individual plans. Agents are able to plan semi-independently subject to the resource requirements of the team, and the need for a separate merging stage is eliminated.

The disadvantage of this approach is that agents are effectively trying to find plans while the external environment (the plans of the other agents) changes around them. Novel techniques are required to plan effectively in these situations.

⁴By analogy to “*non-serialisable*” *subgoals* as identified by Korf (1987).

External summaries In this approach, agents exchange summary information as they plan. The server maintains a shared *external summary* containing summary conditions and temporal constraints from the level 1 nodes in the task trees of each agent⁵. Access to the external summary is controlled with *read and write tokens* and the rules in Figure 4.5, which prevent three types of concurrency issues:

Concurrent writes Data may be corrupted if two agents simultaneously attempt to update the shared summary. Rules 4 and 6 prevent this by queuing write requests so that writes take place serially rather than in parallel.

Dirty reads If one agent is reading the shared summary while another agent is writing to it (or due to write to it), information can be downloaded that is either corrupt or immediately invalidated. Rules 5 and 7 prevent this by forcing agents with read requests to wait if another agent is already trying to update the shared summary.

Avoiding deadlock The combination of rules 7 and 10 prevent the system getting into deadlock. Without rule 10, it is possible to see a situation in which two agents *A* and *B* continually re-issue and re-queue write requests, denying a third agent *C* read access. With rule 10, however, *A* and *B* are required to issue a read request after every write, making sure all agents including *C* get to read the summary before it is updated again.

The avoidance of dirty reads is a contentious issue as preventing agents from reading the shared summary can potentially cause a bottleneck in communication. However, it is important that agents' local versions of the shared summary are as similar as possible when they are considering ways of resolving inter-agent threats, and this is one way of ensuring that this is the case.

Applicability of local search Whenever the external summary information is updated on the server, agents' local copies become invalid. Agents must repeatedly refresh their copies of the external summary to stay up to date with their surroundings. Every time an agent downloads

⁵This blackboard-like approach is equivalent to the broadcasting of external summary information to the team every iteration, but gets around the problem of simultaneous communication.

1. An agent with a *read token* can download copies of the external summary but may not upload new information;
2. An agent with a *write token* can download and upload external summary information;
3. Any number of read tokens may be granted at a time;
4. Only one write token can be granted at a time;
5. Read tokens and write tokens cannot be issued at the same time;
6. Upon requesting a token, an agent is held in a queue until the token can be issued;
7. New read tokens are not issued while an agent is waiting for write permission (the requests, however, are queued);
8. Tokens are issued on a first come, first served basis;
9. Once a token has been issued, it is the agent's responsibility to complete its transaction and release the token as quickly as possible;
10. Each agent is required to perform at least one successful read operation between each pair of consecutive writes.

Figure 4.5: Rules for client-server communication in distributed local planning.

a new external summary, its knowledge of the environment changes. Refinement search algorithms such as those for centralised planning described above rely on a static environment during planning, and are unsuitable for planning in these circumstances without modification.

The advantages of HTN planning and summary information come from abstraction and refinement. While local search through the space of complete plans may seem attractive (Section 2.1.6), this approach is incompatible with the MPF formalism, which imposes a strict structure of decomposition of partial plans. A distributed refinement search is implemented instead, using novel techniques to deal with changes in the external summary. However, local search is an attractive alternative that may be considered in the future (Section 6.2).

Applicability of best first search Best first search, which performs very well in static single agent environments, suffers from several problems in situations where agents are planning socially:

Minimal commitment A key problem arises from the very aspect that makes BFS so fast in single agent problems: its ability to arbitrarily swap between branches of search. Since each agent in a multi agent environment is reliant on its surroundings changing as little as possible, this lack of commitment to a single course of planning can be detrimental to

the team as a whole.

Changing heuristic values The heuristic value of a plan is dependent on inter-plan threats as much as it is intra-plan threats. Heuristic values for stored plans are subject to change whenever the external environment changes. This means that heuristics for open plans need to be frequently recalculated and the open list resorted accordingly. Changing heuristics include `can_any_way` and `might_some_way` as well as quantitative measures of plan quality.

Reopening closed plans Changing heuristic values have dire consequences for systematicity. In single agent planning, plans from the open list may be discarded once they have been processed (Section 2.1.1). In multi agent planning, however, the `might_some_way` value of a pruned plan may change later on in planning such that it becomes a valid plan for further consideration. Consequently, “closed” plans cannot simply be discarded as they can in single agent search.

These problems arise from the independence of the agents in a multi agent team. As an aside, it is possible to exploit concurrency without introducing independence by having a single centralised planning algorithm and allowing agents to “check out” plans from the open list and return results asynchronously (“*distributed global planning*”; Section 2.2.3). In this context, agents become problem solvers within a larger architecture. They do not maintain their own goals and plans, and have limited private internal state. This approach is not a “multi agent” approach in the sense of providing agents with independence during planning, so it will not be considered further.

Applicability of depth first search *Some* of the problems above are dealt with better by depth first than by best first search:

Improved commitment Agents using depth first search make commitments and hold them as long as possible before backtracking. Other agents can make decisions based upon these commitments and be assured a certain level of consistency in their environment (depending on the amount of backtracking required).

Smaller open lists While a list of open plans is maintained, the systematic nature of search means that plans only need to be re-evaluated and resorted when the agent backtracks. Additionally, heuristic orderings are only maintained across the operators resulting from individual refinements, meaning that only a small number of plans need to be reordered each time the agent backtracks.

The problem of reopening closed plans still exists despite these advantages. Other mechanisms are required for achieving completeness and systematicity.

4.4.1 Distributed constraint satisfaction

Inspiration can be taken from *Distributed Constraint Satisfaction Problems (DisCSP)*⁶ and algorithms such as those discussed by Yokoo and Hirayama (2000). This section provides a brief overview of DisCSP and some of the approaches that can be taken.

A *Constraint Satisfaction Problem (CSP)* involves a set of *variables*, each with a set of possible values $domain(var \in variables)$ and a set of *constraints* defined on tuples of variables. For the purposes of this discussion all constraints are assumed to be binary. The aim of the problem is to find a set of variable assignments for which all of the constraints hold. An example problem is shown in Figure 4.6.

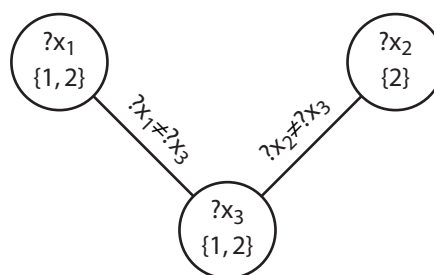


Figure 4.6: Example constraint satisfaction problem.

A DisCSP is a variant of a CSP where each variable is owned by a separate agent. Agents communicate by passing *messages* to one another. Messages take a finite time to arrive but

⁶The abbreviation *DCSP* is typically used to refer to *dynamic constraint satisfaction problems*, which are very different, and are not covered here.

guaranteed to arrive in the order they are sent. Each agent has a *neighbourhood* of other agents with which it can interact, defined by the connectivity of the local constraints defined on its variable.

Each agent has an *agent_view* made up of the variable assignments of its neighbours. Agents are concerned with making assignments to their own variables that are consistent with local constraints and their agent view.

DisCSP versus multi agent planning Plans in multi agent planning are similar in many ways to variables in DisCSP⁷:

- in planning, partial plans are owned by separate agents; in DisCSP, variables are owned by separate agents;
- in planning, partial plans represent disjunctions of possible histories; in DisCSP, variables represent disjunctions of possible values;
- in planning, agents are trying to find compatible sets of histories; in DisCSP, agents are trying to find compatible sets of values.

The domains of variables in DisCSP are small and finite. The closest equivalent to an individual value in multi agent planning is a *history* (Section 3.2.2): a completely decomposed, grounded and ordered plan. According to this model, the domain of a partial plan may be very large, and in some planning mechanisms infinitely so. The mechanisms from Chapter 3 put a maximum limit on the depth of recursion, making problems in MPF potentially large but finite in size.

Similarly, DisCSP agents can make and retract variable assignments quickly and without much effort. This is in sharp contrast to multi agent planning, where the production of a grounded plan can be time consuming and may take many iterations of the planning algorithm.

Agent views provide efficiency through localisation in DisCSP because collections of variables may only be sparsely interrelated. Plans in multi agent planning may have many preconditions and effects, and in “traditional” academic planning problems these provide lots of interactions. This makes the potential for inter-plan conflict great and the benefit from localised agent views

⁷They are more similar in many ways to sets of variables, but this would imply that a single plan is owned by several agents.

rather small. Whether DisCSP is faster or slower than CSP depends on the structure of the problem, the number of variables, the distribution of the agents responsible for them and the density of intra- and inter-agent constraints.

Agents in DisCSP are only interested in finding conflict free values for their own variables, but they obey a standard distributed search algorithm. This makes them self interested but trustworthy from the point of view of the definitions in Section 1.1.4.

Success requires that there is a set of variable assignments that satisfies all the constraints specified in the problem. In this is not the case, the algorithms presented here will fail. However, with appropriate negotiation techniques it may be possible for agents to agree on a subset of constraints that *can* be satisfied (Section 1.1.3).

Asynchronous Backtracking (ABT) Yokoo and Hirayama discuss several algorithms for the solution of DisCSPs, the most relevant of which is *Asynchronous Backtracking (ABT)*. This algorithm is provably sound and complete, although it involves no maintenance of open lists. The key properties of ABT are as follows:

Agent priorities To avoid infinite loops, agents are arbitrarily assigned *priorities*. Priorities are fixed for the duration of problem solving. Lower priority agents have to submit to higher priority agents whenever a conflict occurs. When an agent is happy with its solution it keeps it until a joint solution is found or an alternative assignment is demanded by an agent of higher priority.

Message passing Search is controlled by the passing of messages between neighbouring agents. There are two types of message. Whenever an agent assigns a value to its variable, it sends an *okay?* message to each of its neighbours, communicating the change and checking whether it violates any constraints. Neighbours of higher priority can veto the assignment by responding with *nogood* messages indicating sets of assignments that will not work together.

Nogood constraints To prevent the system getting stuck in local minima or plateaux of heuristic value, *nogood* constraints are added whenever *nogood* messages are received. These extra constraints act just like the initial constraints in the problem specification (Figure

4.6), preventing the agents making the same conflicting assignments twice. Nogood constraints ensure completeness as they progressively restrict the search space until a solution is found or the useful domain of one or more agents' variables becomes empty.

Other constraint satisfaction algorithms The ABT algorithm uses a statically defined priority order to prevent loops. This can be restrictive, however, if the agent with the most constraining part of the problem has a low priority. A second algorithm, *Asynchronous Weak Commitment (AWC)*, allows agents to increase their own priorities whenever they are unable to find a valid value for their variable. This allows poor decisions to be revised without extensive repeated search.

A third algorithm, *Distributed BreakOut (DBO)*, assigns a real valued *weight* to each constraint. The sum of weights of potentially conflicting constraints is used as a heuristic when choosing values for assignments. Agents work together in pairs to try to resolve conflicts: the agent that is able to minimise the total heuristic value of the pair is given permission to change the value of its variable. Weights are increased whenever a pair share violated constraints and cannot improve on their current situation: this shapes search by indicating which constraints are the most important in terms of how much they restrict the set of possible solutions.

4.4.2 Distributed planning with DisCSP techniques

The maximum depth imposed on recursion in Section 3.6.1 means that a maximum bound is known for the number of selectable decompositions, orderable tasks and bindable variables. This means that any multi agent planning problem expressed in MPF can be rewritten as a constraint satisfaction problem. However, there are arguments for maintaining a planning specific representation rather than encoding it as a different form of problem (Brafman and Hoos, 1999), including the use of more “natural” definitions of planning oriented refinements, operators and heuristics. The distributed local planning algorithm, shown in Figure 4.7, uses novel ways of approximating some of the features of ABT and its sister algorithms without encoding problems as DisCSPs. MPF is used as the basic search mechanism. The main features of the algorithm are discussed below:


```

1  function find_plan(initial) → solution:
2      create empty open_list
3      open_list ← push(copy(initial), open_list)
4      while ¬ empty(open_list):
5          handle_messages(open_list)
6          (plan, open_list) ← pop(open_list)
7          sync(plan)
8          if ¬ might_some_way(plan):
9              send_messages(plan)
10         else if can_any_way(plan):
11             solution ← plan
12             return
13         else:
14             ref ← pick_refinement(plan)
15             if ref ≠ null:
16                 open_list ← push(plans(ref), open_list)
17     solution ← failure

18 function sync(plan):
19     request and wait for write token
20     download external and update plan
21     if might_some_way(plan):
22         upload revised external from plan
23     release write token

```

Figure 4.7: Algorithm for distributed local planning. The `handle_messages` and `send_messages` functions are shown in Figure 4.8.

External summaries Communication is an essential feature of any decentralised planning algorithm capable of solving problems like the “airlock” problem in Figure 1.2. Agents must share information about their plans if they are to make useful decisions about choices of refinements and operators. Summary information is useful in this respect because it conveys lots of information about potential resource usage in a compact format for communication. Agents broadcast summary information each time they change their current plan, using the *external summary* mechanism described above.

Summary weighting In CSP, *nogood constraints* (or *nogoods* for short) represent combinations of values that cannot be chosen, for example:

$$\neg(x_1 = 1 \wedge x_2 = 1 \wedge x_3 = 3)$$

Ideally nogoods represent minimal sets of incompatible assignments. However, the sets of conflicting constraints discovered during problem solving are often not minimal, and minimal subsets can be time consuming to calculate. CSP and DisCSP algorithms normally allow the storage of non-minimal nogoods simply in the interests of speed.

In planning, nogoods correspond to sets of incompatible decompositions, orderings and binding constraints. Since the set of combinations of these constraints is very large, the calculation and storage of such fine grained nogoods is impractical: an alternative form is needed.

Another interpretation of a nogood is a set of threats that cause a plan to fail. This basically involves any combination of `must_clobber` relationships in a `¬might_some_way` plan. This kind of relationship is typically only uncovered late on in planning when tasks have been decomposed to a fine level of detail, so typically only threats at low levels of abstraction can be detected. One problem with summary conditions is that, because they are merged during summarisation (Section 3.3.9), they are different at every level of abstraction. This means that abstracted nogoods, or `may_nogoods`, must be calculated between every combination of ancestors of the summary conditions involved. The generation and checking of this information is prohibitively costly: a simpler representation is needed.

“Soft” constraints that alter the heuristic plan quality function may be used as an alternative to nogoods. One way of implementing soft constraints is to use a weighting system on nodes in the task tree, similar to the weighting system in distributed breakout above. Each summary condition and state constraint is given an extra “weight” field. The weights of summary conditions of state constraints involved in conflicts are increased whenever an agent backtracks, and the weights of summary conditions of primitive and abstract tasks are derived from them:

- weights are *maximised* when summarising task networks because summary conditions at task networks represent conjunctions of the summary conditions of their children;
- weights are *minimised* when summarising tasks because summary conditions at tasks represent disjunctions of the summary conditions of their children.

This requires some summary information to be recomputed when weights are changed, but does not require the storage of multiple annotations per summary condition. Summary weighting is of low cost to planning agents compared to the approaches described above, but has some limitations:

1. Weights only *suggest* which refinements and operators to select during planning; they do not prune branches of search. This means that weighting does not provide completeness like nogoods do.
2. Nogoods are very specific about which sets of constraints are allowed and which are not. A complete nogood data structure is a disjunction of disallowed conjunctions of variable assignments. Weights are much less specific. In particular there is no disjunctive part to the weighting system: weights are considered all at the same time rather than in associated groups.
3. Because individual summary conditions are weighted rather than complete links, threats other than those originally intended may be adversely affected.

Third party conflicts ABT in DisCSP does not explicitly consider situations in which a constraint between two agents can only be satisfied by a third party. For example, if a postcondition of a task $task_1$ owned by an agent $agent_1$ clobbers a precondition of a task $task_2$ owned by an agent $agent_2$, it may be that the only way of breaking the relationship is to *block* it with a third task $task_3$. Two possible situations result:

1. If $task_3$ is primitive and grounded, it may be directly ordered as a blocker using the *separation* or *exclusion* strategies described in Section 3.3.8.
2. If $task_3$ is abstract or lifted, it might only *may_block* the threat between $task_1$ and $task_2$. In this case $task_3$ must be decomposed and/or grounded appropriately to be used as a blocker.

If $task_3$ is not owned by $agent_2$ in the second of these situations, $agent_2$ will be unable to perform the necessary decomposition to ensure that blocking occurs. There are two ways around this:

Decomposition requests $agent_2$ can send a message to the owner of $task_3$ requesting that it decomposes the task. This is a non-trivial operation as several refinements may be necessary to decompose $task_3$, and there may be several ways of performing the full decomposition. The two agents will have to agree on the best choice of decomposition, either by heuristic estimates or by some form of negotiation.

Decomposition-then-ordering $agent_2$ can simply avoid order operations until it can be sure third party decomposition will not be required. When picking refinements, agents avoid any `order_ref` where there are external `may_achieve` relationships with a source node with `existence = may`. `decompose_ref` refinements are always allowed because they are unaffected by causal links and threats. As the abstract tasks in the set of individual plans are converted to primitive tasks, more `order_ref` refinements become possible. If an agent cannot pick a refinement at any point it simply waits until one becomes available.

The distributed local planning algorithm described in Figure 4.7 uses the second of these approaches. It uses a combination of message passing, summary weights and decomposition-then-ordering to simulate the asynchronous backtracking approach to DisCSP.

Message passing Whenever an agent is forced to backtrack, it checks to see if the backtracking was caused by any inter-plan threats. If this is the case, it sends two types of message to the other agents involved:

- penalise messages cause agents to update the weights of relevant leaf nodes in their task trees to reflect the threats.
- reset messages cause agents to backtrack to their *initial* plans.

The message sending and handling functions are shown in Figure 4.8. The `send_messages` function is invoked when an agent is about to backtrack. If any inter-plan threats are discovered, penalise and reset messages are sent such that all agents involved (including the originating agent) restart their searches with new weights. If no inter-plan threats are identified, simple backtracking is used and no messages are sent.

```

1  function send_messages(plan):
2      for each threat  $sum_1 \xrightarrow{must\_lobber} sum_2$ :
3          create set to_reset  $\leftarrow \emptyset$ 
4          if  $owner(node(sum_1)) \neq owner(node(sum_2))$ :
5              send penalise(sum1) message to  $owner(node(sum_1))$ 
6              send penalise(sum2) message to  $owner(node(sum_2))$ 
7              to_reset  $\leftarrow to\_reset \cup \{owner(node(sum_1)), owner(node(sum_2))\}$ 
8      for each agent  $\in to\_reset$ 
9          send reset message to agent

10 function handle_messages(msg):
11     if there are any waiting reset messages:
12         plan  $\leftarrow initial$ 
13     for each waiting penalise message msg:
14         penalise(plan, msg)
15         penalise(initial, msg)

```

Figure 4.8: Algorithms for message passing in distributed local planning.

Waiting It is possible for an agent to reach a state where it is waiting on other agents in the team to be able to do anything. This occurs in two situations:

- an agent has a `can_any_way` plan, but cannot upload a solution because other agents are still planning.
- an agent has a completely decomposed `might_some_way` plan, but cannot perform any refinements because other agents are yet to make necessary decompositions.

In these situations the agent simply waits until a suitable option becomes available. Assuming the other agents remain planning and in communication, an option will eventually arise or a reset will be forced.

Agent independence Distributed local planning is the most sensitive of the approaches presented in this chapter to the independence of the planning agents. Agents never exchange more information than is necessary for the coordination of their plans. They also maintain control of their own plans throughout the entire planning process. They are self-interested, but have to be trustworthy and follow the planning algorithm: this makes negotiation unnecessary when goals are compatible. Like centralised planning and plan-then-merge, however, distributed local

planning only works when goals are compatible and agents are not competitive. As with the other approaches, negotiation is needed if subsets of goals have to be selected to find a partial solution.

4.5 Summary of planning algorithms

Three planning algorithms have been presented in this chapter, implementing the three multi agent planning approaches identified in Section 2.2 using the common planning mechanism developed in Chapter 3:

Centralised planning draws all agents' individual problems together into a single joint planning problem that is solved by a single agent. This is bad from an independence point of view, but allows the use of conventional refinement planning techniques that are hopefully efficient, systematic, sound and complete.

Plan-then-merge is an implementation of a class of *plan merging* approaches. Agents create individual plans without considering each others' goals, knowledge, capabilities and so on, and pass them to a single agent for merging into a coordinated joint plan. Planning and merging agents use similar refinement planning algorithms to the agent in centralised planning (the plan merging algorithm is simply a variant of the planning algorithm), but are allowed partial independence in that goals and some knowledge can be kept private.

The partial parallelisation of planning in plan-then-merge may make it faster than centralised planning in some cases, but the costs of the extra plan merging stage may prevent this in other cases. In addition, the division of search into discrete, non-overlapping areas may prevent plan-then-merge from being able to solve some kinds of problem.

Distributed local planning removes the plan merging stage of plan-then-merge. Agents are allowed to exchange summary information during planning to achieve the coordination that would have been achieved by plan merging. Standard refinement planning algorithms are inappropriate in this situation as every agent is essentially planning subject to a changing environment. Agents use a hybrid planning algorithm instead that is inspired by refine-

ment planning and distributed constraint satisfaction. Unfortunately, due to the nature of its implementation, this novel algorithm is not guaranteed to be sound or complete.

In the next chapter, these approaches are compared on a number of “traditional” HTN planning problems using a number of criteria. Chapter 6 concludes the thesis by revisiting and analysing the contributions outlined in Section 1.3 and suggesting possibilities for future work.

Chapter 5

Experiments and empirical analysis

This chapter contains an empirical analysis of the planning mechanisms and algorithms described in Chapters 3 and 4. The two planning mechanisms and the three planning algorithms were compared on a variety of planning domains using a number of criteria. The criteria and domains are described in Sections 5.1 and 5.2 respectively, and the experimental results are discussed in Sections 5.3 and 5.4.

5.1 Evaluation criteria

Evaluation criteria are an essential part of empirical analysis as they provide the measure by which to assess the success or failure of an approach. Different criteria were used when evaluating the planning mechanisms from Chapter 3 and approaches from Chapter 4.

It should be noted that the algorithms presented in this thesis are not competitive when compared against current single agent planners. The purpose of this thesis is not to compete on such a level, but rather to highlight the relationships between different approaches and features of problems. The important consideration is the relative performance of the algorithms when applied to relevant problems (Section 1.2): if one algorithm is consistently poorer than the others, for example, an analysis of the discrepancy may lead to a better understanding of how to construct future related approaches.

Planning mechanisms The pMPF and fMPF mechanisms were compared using four criteria:

Tree size Smaller task trees place smaller memory requirements on search algorithms and speed up basic bookkeeping tasks such as creating copies of plans and searching for specific nodes and constraints. Tree size may be measured in several ways: height, number of nodes, number of planning variables, and so on.

Tree generation time The creation of an initial task tree is a non-trivial part of the complete planning process. Smaller trees of an equivalent mechanism will obviously be faster to create, but the addition of planning variables to fMPF trees add a layer of complexity not present in pMPF, increasing generation time per node.

Quality of summary information The production of an initial task tree is only a small part of the overall planning process. The quality of summary information generated by the tree will have a great effect on the quality of threat based heuristics and planning performance. One concern is that fMPF may provide less accurate summary information than pMPF.

Planning algorithms The centralised planning, plan-then-merge and distributed local planning algorithms were compared using five criteria:

Ability to solve problems Weaknesses in the decentralised planning approaches make them unable to solve certain classes of problem, even when the problems can be represented using MPF. For example, if a planning agent in plan-then-merge is unable to find a plan for its individual subproblem, the whole team fails as a consequence.

Solution efficiency An algorithm that is technically capable of solving a problem can sometimes still fail because it is unable to find a solution within the available time and/or memory. This is especially relevant to local search algorithms like distributed local planning where completeness is not guaranteed.

Plan quality When a plan has been found there is no guarantee it will be an optimal solution to the problem. Because the problems tackled in in this chapter are small, the plans produced can be compared with hypothetical optimal plans. Plan quality is measured in terms of the number of tasks in a plan: the fewer tasks in a correct solution plan, the better.

Agent independence As discussed in Chapter 1, multi agent planning in general incorporates a wide variety of problems: much larger and more complex than the problem subset chosen for analysis here. In general “real world” problems, agent independence is very important. Agents should be able to plan and coordinate their plans independently, without sharing information with or relying on the services of others, to maintain robust performance and a competitive advantage in a variety of problems.

Independence has been discussed in previous chapters and will be revisited in Chapter 6: it will not be discussed further in this chapter.

5.2 Experimental domains

Three test domains were used as the basis of empirical analysis. Two of these, *Blocksworld* and *Navigation*, are examples of recursive planning domains. Blocksworld obeys the action independence assumption (Section 3.6.1) whereas Navigation does not. A third, *Holes*, is a non-recursive domain primarily involving choices of resource. These domains were chosen because they represent different types of problem: Blocksworld gives all agents complete freedom to sense and alter the world, Navigation imposes restrictions on the robots that agents can control, and thus the state literals they can change, and Holes imposes further restrictions on the types of state information they can sense. The domains are described in more detail below.

Only problems with valid solutions are considered. The performance of the algorithms on unsolvable problems is well known: centralised planning and plan-then-merge fail after an exhaustive search of the problem space, and distributed local planning fails after a predefined time limit or number of iterations. The solution of problems with conflicting goals requires extra negotiation techniques that select a compatible subset for solution: this will be revisited in Chapter 6.

5.2.1 The Blocksworld domain

The Blocksworld variant used here is a typed implementation the HTN Blocksworld presented by Erol (1996). World objects are organised into two types, *blocks* and *tables*, subsumed

under a third type, *surfaces* (grippers are not explicitly modelled). World state is described using two sets of predicates: $clear(?surface)$ and $on(?block, ?surface)$. Two abstract tasks, $achieve_on(?block, ?surface)$ and $achieve_clear(?block)$, are used in level 1 of the task tree, with a primitive task, $move(?block, ?surface_1, ?surface_2)$, used for their implementation. A complete domain description and sample problem are given in Section B.1.

$achieve_on$ is defined in terms of $achieve_clear$ and $move$. $achieve_clear$ is recursive, being defined in terms of itself and $move$. An initial task subtree for a level 1 $achieve_clear$ task contains possible $move$ tasks for all of the other blocks in the problem: the planning agent effectively has to identify the blocks that are currently on top of the target block and choose alternative locations for them. These *moves* effectively remove all possible preconditions from the $achieve_clear$ task: the subtree of the task will contain appropriate histories for clearing the block regardless of the initial conditions of the problem (Section 3.6). Consequently, $achieve_on$ and $achieve_clear$ both obey the action independence assumption. Blocksworld task trees may therefore be reliably generated using the value counting technique from Section 3.6.1.

Four types of Blocksworld problem were used:

bwUnstack problems These single agent problems¹ involve clearing the bottom block in a tower. Problems are named $bwUnstack_x$ where x is the number of blocks in the initial tower. Each problem involves a single level 1 $achieve_clear$ task. This makes the problems useful for comparing task tree sizes and generation times for the pMPF and fMPF mechanisms with different numbers of values of free variables (Section 3.5).

bwSwap problems These problems involve reversing the order of lots of pairs of blocks. Problems are named $bwSwap_x$ where x is the number of blocks: there are $x/2$ towers in each problem. The level 1 tasks in these problems are *independent* of each other (Korf, 1987): each task swaps a pair of blocks that are not involved in any other tasks, so inter-task conflicts do not arise in sensible solution plans.

¹All the other problems presented, in this and other domains, apply to single and multiple agents.

bwReverse problems These problems involve reversing the order of a tower of blocks. Problems are named *bwReverse_x* where x is the number of blocks in the problem.

In these problems the overall goal to reverse the tower of blocks is specified as a set of *achieve_on* tasks, each of which is broken down into three parts according to the specification of the *achieve_clear* method in the domain description (Section B.1):

1. Clear the block x to be moved.
2. Clear the destination block y .
3. Perform the move $x \rightarrow y$.

Step 3 above is a simple primitive task. Steps 1 and 2 involve recursively clearing all blocks on top of x and y . It is these tasks that cause the bulk of the work for the planner. Given a set of level 1 *achieve_on* tasks for building a tower, a number of duplicated level 3 *achieve_clear* tasks are created as shown in Figure 5.1.

Level 1 task	Level 3 tasks
$achieve_on(A, B)$	\rightarrow $achieve_clear(A)$ $achieve_clear(B)$
$achieve_on(B, C)$	\rightarrow $achieve_clear(B)$ $achieve_clear(C)$
$achieve_on(C, D)$	\rightarrow $achieve_clear(C)$ $achieve_clear(D)$

Figure 5.1: Redundancy in level 3 *achieve_clear* tasks in *bwReverse_4*. Pairs of duplicated tasks are shown in blue and orange.

Once decomposed, these level 3 tasks will create other duplicate *achieve_clear* tasks. The planning agent(s) must decide which tasks to keep and which to discard using the *achieve_clear_noop* method (Appendix B.1).

The overall complexity of any multi agent planning problem is partly determined by the distribution of tasks between the agents, and *bwReverse* problems are a good example of this. Two variants of the *bwReverse* problems were run:

bwReverseRobin problems When the server distributes initial plans, it assigns *achieve_on* tasks to agents starting at the top of the initial tower and working towards the bottom. In *bwReverseRobin* problems, tasks are assigned in a round robin sequence. For example, the tasks for a tower of five blocks would be distributed between two agents as follows:

$$\begin{aligned} \text{achieve_on}(A, B) &\rightarrow \text{agent}_1 \\ \text{achieve_on}(B, C) &\rightarrow \text{agent}_2 \\ \text{achieve_on}(C, D) &\rightarrow \text{agent}_1 \\ \text{achieve_on}(D, E) &\rightarrow \text{agent}_2 \end{aligned}$$

bwReverseSeq problems In these problems, tasks are assigned to agents in sequence such that each agent receives a set of tasks for adjacent blocks in the tower. For example, the tasks above would instead be distributed as follows:

$$\begin{aligned} \text{achieve_on}(A, B) &\rightarrow \text{agent}_1 \\ \text{achieve_on}(B, C) &\rightarrow \text{agent}_1 \\ \text{achieve_on}(C, D) &\rightarrow \text{agent}_2 \\ \text{achieve_on}(D, E) &\rightarrow \text{agent}_2 \end{aligned}$$

bwRandom problems These are randomly generated Blocksworld problems, named *bwRandom_x_y*, where x is the number of blocks involved and y is an index identifying the randomly generated initial conditions and goals.

5.2.2 The Navigation domain

Navigation is an implementation of the multi-robot navigation domain introduced in Section 3.6.1. World objects fall into two categories: *robots* and *locations*. World state is described using three sets of state literals: $\text{edge}(?location_1, ?location_2)$, $\text{at}(?robot, ?location)$ and $\text{clear}(?location)$. A single recursive abstract task, $\text{travel}(?robot, ?location_1, ?location_2)$, is used in level 1 of the task tree, defined in terms of itself and a primitive task, $\text{move}(?robot, ?location_1, ?location_2)$. Preconditions and effects in the domain are structured such that two robots cannot occupy the same location or travel on the same edge at the same time. A complete domain description and sample problem are given in Section B.2.

As discussed in Section 3.6.1, the ability to move from one location to another depends partly on the position of other robots on the location graph. Because subtrees of *travel* tasks only allow the control of a single robot, this domain does not obey the action independence assumption

when more than one robot is present.

Three types of Navigation problems are used in the following experiments:

navLine problems These problems involve moving robots along a line of locations. Problems are named $navLine_{x-y}$ where x is the number of locations and y is the number of robots. As shown in Figure 5.2, robots are moved “in convoy” from one end of the line to the other.

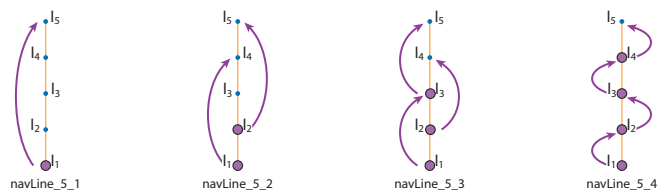


Figure 5.2: Topology of $navLine_{5-1}$ to $navStar_{5-4}$. Blue dots represent locations, orange lines represent edges, purple circles represent the starting locations of agents, and purple arrows point to goal locations.

navRing problems These multi robot problems involve moving robots on a ring of locations. Problems are named $navRing_{x-y}$ where x is the number of locations in the ring and y is the number of robots. Robots are initially spaced equally around the ring. Each robot must travel to the location on the ring that is farthest from its initial starting point.

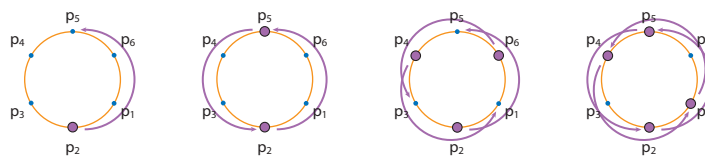


Figure 5.3: Topology of $navRing_{6-1}$ to $navRing_{6-4}$. Symbols have the same meanings as those in Figure 5.2.

navStar problems These multi robot problems involve moving robots on a star-shaped graph. Each “point” on the star consists of two locations as shown in Figure 5.4. Problems are named $navStar_{x-y-z}$ where x is then number of points on the star² and y is the number of robots.

²There are $2x + 1$ locations on the star in total.

Problems are generated with random start/destination points for the robots: z is a number identifying the pseudo-random configuration of start points and destinations problem.

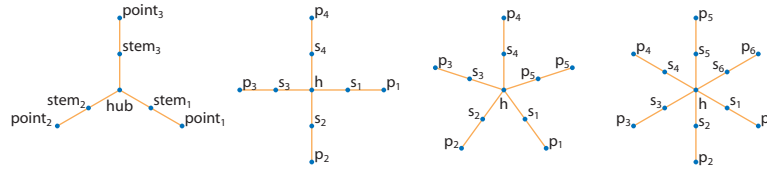


Figure 5.4: Topology of *navStar_3_y* to *navStar_6_y*. Blue dots represent locations and orange lines represent edges. Starting and goal locations are randomly determined for each problem, so they are not shown.

Although Navigation does not obey the action independence assumption, all of these Navigation problems may be successfully generated and solved using value counting (Section 3.6). However, this is not the case for Navigation problems in general, as was discussed in Section 3.6.1 (Figure 3.35).

5.2.3 The Holes domain

Holes is a non-recursive domain inspired by the children’s game where different shaped pegs must be placed into holes cut into a wooden board. World objects fall into three categories: *pegs*, *holes* and *features*. The set of features may be subdivided into *shapes*, *colours* and *sizes*. Blocks and holes are subsumed by a super-type *objects*. World state is described by five sets of predicates: $in(?peg, ?hole)$, $empty(?hole)$, $shape(?object, ?shape)$, $colour(?object, ?colour)$, and $size(?object, ?size)$. A complete domain description and sample problem are given in Section B.3.1.

Problems involve inserting pegs into holes. This can only be done for combinations of peg and hole with the same set of features, as described by the relevant *shape*, *colour* and *size* predicates. More than one shape, size or colour can be specified per hole, allowing pegs of various kinds to be placed there.

Pegs are placed using the abstract action $locate(?peg)$, which is implemented with a number of methods, a subset of which may be available to an agent:

- The method *match_all* chooses a hole with a matching shape, colour and size and places the peg there.
- The method *match_shape* chooses a hole with a matching shape and places the peg there.
- The method *match_color* chooses a hole with a matching colour and places the peg there.
- The method *match_size* chooses a hole with a matching size and places the peg there.
- The method *verify_shape* does nothing but checks that a peg placed with *match_color* or *match_size* has been located successfully according to its shape.
- The method *verify_color* does nothing but checks that a peg placed with *match_shape* or *match_size* has been located successfully according to its colour.
- The method *verify_size* does nothing but checks that a peg placed with *match_shape* or *match_color* has been located successfully according to its size.

These methods are implemented in terms of pre- and postconditions involving the relevant feature literals and tasks involving a primitive action *place(?peg, ?hole)*, which updates the relevant *in* and *empty* literals. Methods are selectively assigned to agents to create two types of problems:

holesGeneral problems In these problems, agents are given access to the *match_all* method. Each agent has one or more pegs to place. Problems are randomly generated such that each hole is specific about the values of certain feature types and relaxed about others. For example, a hole might accept red pegs of any size or shape, or large, round pegs of any colour. Each hole in a problem is specific about the same number of features, although the features themselves are chosen on a per-hole basis. Problems are named *holesGeneral_w_x_y_z* where *w* is the number of pegs and number of holes, *x* is the number of features about which each hole is specific, *y* is the number of values per feature, and *z* is an index identifying the random assignment of features for the given values of *w*, *x* and *y*.

holesSpecial problems Three agents are present (one agent in centralised planning). The first agent is given access to the *match_shape* and *check_shape* methods, the second is given access to the *match_color* and *check_color* methods, and the third is given access to the *match_size* and *check_size* methods. Each agent is given the task of placing *all* the pegs in the correct

holes. Because agents are only able to reason about one type of feature each, they are forced to cooperate to identify which holes are appropriate. Problems are named *holesSpecial_w_x_y_z* where w , x , y and z have the same meanings as those for *holesGeneral* problems. In particular, the same random assignment of features is represented by a given combination of values of w , x , y and z in each type of problem.

While they appear similar and have similar names, *holesGeneral* and *holesSpecial* problems are vastly different in terms of complexity. *holesSpecial* problems involve much larger numbers of tasks and potential conflicts, greatly increasing the size of task trees and search space alike.

5.3 Comparison of planning mechanisms

The pMPF and fMPF mechanisms (Sections 3.3 to 3.6) require empirical comparison. It is clear that first order task trees will have the same number or fewer nodes than equivalent compiled propositional task trees. In fact, first order trees avoid an exponential explosion in the number of nodes on each level relative to the number of relevant world objects in the problem (Section 3.5). However, the effects of tree type on planning demand investigation.

Table 5.1 shows tree generation and planning data for for various *bwUnstack* and *navLine* problems, chosen because they are recursive and have predictable structure. Each figure in the table is averaged over five runs of the relevant algorithm. 10 minutes and 512 MB heap space were allowed during tree generation. Where trees were successfully produced, a further 10 minutes and 512 MB heap space were allowed to create a solution plan using centralised planning. A run was considered a failure if it exceeded either of these limits, although runs that timed out are marked differently than runs that explicitly failed in tables of results.

Tree size The heights of the task trees in *bwUnstack* and *navLine* problems is proportional to the number of values of the key variables in the problems. However, the sizes of the trees are very different in terms of the number of nodes they contain:

- In *bwUnstack* problems there is a single level 1 task. The task has a subtree of height

Problem	First order (fMPF)					Propositional (pMPF)						
	generation time	nodes vars			planning time	iterations	generation time	nodes vars			planning time	iterations
		nodes	vars					nodes	vars			
<i>bwUnstack_2</i>	50	34	14	224	1	151	19	0	184	1		
<i>bwUnstack_3</i>	116	61	27	529	3	458	75	0	446	2		
<i>bwUnstack_4</i>	153	88	40	1173	4	1565	439	0	1277	3		
<i>bwUnstack_5</i>	185	115	53	2340	5	38388	3751	0	5231	4		
<i>bwUnstack_6</i>	344	142	66	4160	6	300769	42967	0	t/out (p)			
<i>bwUnstack_7</i>	347	169	79	8229	7	t/out (t)						
<i>bwUnstack_8</i>	329	196	92	14669	8	t/out (t)						
<i>navLine_3_1</i>	107	62	32	636	2	241	126	0	529	2		
<i>navLine_4_1</i>	145	86	45	1625	3	1251	742	0	1449	3		
<i>navLine_5_1</i>	202	110	58	3097	4	10717	6486	0	7780	4		
<i>navLine_6_1</i>	337	134	71	6311	5	124790	75524	0	t/out (p)			
<i>navLine_3_2</i>	227	116	64	678	3	457	244	0	720	3		
<i>navLine_4_2</i>	209	161	90	4711	6	2584	1473	0	2450	6		
<i>navLine_5_2</i>	235	206	116	9304	9	23880	12958	0	6441	36		
<i>navLine_6_2</i>	660	251	142	34515	18	t/out (t)						
<i>navLine_4_3</i>	399	236	135	1204	5	3905	2204	0	3696	5		
<i>navLine_5_3</i>	467	302	174	7067	10	53672	19430	0	183915	121		
<i>navLine_6_3</i>	842	368	213	32133	18	t/out (t)						
<i>navLine_5_4</i>	813	398	232	2082	7	87275	25902	0	105784	7		
<i>navLine_6_4</i>	730	485	284	19634	14	t/out (t)						

Table 5.1: Tree generation and planning data for various recursive problems. All times are in milliseconds. Experiments that exceeded time or memory limits during tree generation are marked *t/out (t)*. Experiments that exceeded time or memory limits during planning are marked *t/out (p)*. Figures for planning that were not collected due to failure during tree generation are left blank.

$O(x)$, where x is the number of blocks. The branching factor of the tree discounting state constraints is $O(x)$ for propositional task trees and 1 for first order trees.

- In *navLine* problems there are y level 1 tasks, where y is the number of robots in the problem. Each task has a subtree of height x , where x is the number of locations. As in *bwUnstack* problems, the branching factor is $O(x)$ for propositional task trees and 1 for first order trees.

The task trees in Table 5.1 show this pattern: first order trees have $O(x)$ nodes with respect to the number of blocks/locations in the problem, while propositional nodes have $O(x^x)$.

Task tree size in general is primarily determined by the abstract task and method with the highest branching factor. In domains with mutually recursive tasks and methods, however, the situation can be far worse. In these situations, tree size is dependent on the branching factors of all tasks and methods involved in the recursion. Blocksworld and Navigation, for example, are both singly recursive domains: each recursive method only calls itself once. The cost tree generation and summarisation increases exponentially when doubly recursive methods or worse

are introduced.

Consider, for example, a planning domain in which an agent has to retrieve data from a binary tree. An abstract *scan_node(?node)* task would be implemented with three methods, one returning the value stored at *?node*, one searching the left hand branch and one searching the right hand branch. Both the left and right methods would be implemented in terms of other *scan_node* tasks. assuming value counting is used in task tree generation, the number of nodes in a first order task tree for a problem in this domain would be $O(2^x)$, while the number of nodes in a propositional tree would be $O((2x)^x)$.

Tree generation time Tree generation time in the problems above is roughly proportional to the size of the tree (a few milliseconds per node). The ratio of time to size is similar for both first order and propositional task trees. It is primarily the number of nodes in the tree that affects generation time, so propositional trees are at a severe disadvantage.

Summary information and planning times While it is not shown here, the summary information produced by initial first order and propositional task trees is identical. In simple problems, the number of planning iterations is similar for the two mechanisms: any small discrepancies (for example, the difference in number of planning iterations in *bwUnstack_3* to *bwUnstack_5*) are artefacts of tree simplification (Section 3.4.4) and become negligible as the recursion depth increases. However, two results indicate that a large discrepancy in planning iterations might come about as problem complexity increases: in *navLine_5_2* and *navLine_5_3*, pMPF takes many more iterations than fMPF. A hypothesis for this is as follows:

Initial pMPF and fMPF task trees produce identical summary information, and on the surface, refinements and planning operators in pMPF and fMPF are very similar. However, the decomposition refinement in fMPF allows more flexibility than that in pMPF because of the way it handles *free variables* from methods (Section 3.5).

In pMPF, free variables in methods are bound to world objects during task tree generation. When a pMPF planner decomposes a task, it chooses a fully grounded decomposition and is

stuck with the bound variables therein: backtracking is required if the bindings turn out to be incorrect. In fMPF, however, free variables are added to the task tree in their unbound state. The presence of free variables generally has a positive effect on planning performance, because fMPF planners can postpone variable binding until long after decomposition, when the plan is generally less abstract and more accurate summary information is available. This is in agreement with the principle of least commitment (Section 2.1.4) because binding of free variables is delayed until it is required to resolve threats. However, delayed binding can have a negative effect in complex problems where there are many potential bindings (Section 5.4.3), as lots of summary information has to be recalculated after each possibility is tested: this is a costly process when individual planning variables are used widely in the task tree.

The necessity of generating and maintaining a task tree clearly limits the scalability of MPF. However, as predicted in Section 3.5, the use of first order task trees alleviates some of the problem, allowing the use of smaller task trees for problems of the same complexity. Consequently, fMPF is chosen as the mechanism for the experiments in the remainder of this chapter.

5.4 Comparison of planning approaches

This section compares the performance of the planning approaches from Chapter 4 on the problems described in Section 5.2 above. Tables of results are included for each type of problem examined. The tables have a common structure, described below. The reader is referred to Table 5.10 for concrete examples of the various labels described.

Results are labelled according to the planning algorithm and number of agents involved:

- *c* indicates centralised planning;
- *mn* indicates plan-then-merge with *n* agents;
- *dn* indicates distributed local planning with *n* agents.

In most experiments, seven algorithms are compared:

- centralised planning;

- plan-then-merge with 2 to 4 agents;
- distributed local planning with 2 to 4 agents.

However, as the maximum number of agents in a problem is the same as the number of level 1 tasks in the initial task tree, some experiments were restricted to smaller teams of agents.

Each result is averaged over five runs of the relevant algorithm. Runs are limited to 10 minutes maximum running time, 2 minutes per iteration and 512 MB total heap space. Two numbers are listed where one or more runs was a success:

- the fraction of runs that succeeded;
- the average solution time for successful runs.

Solution times for plan merging are calculated as the sum of the average planning time and the average plan merging time. The times listed are exclusive of the time taken to download initial task trees and upload solutions (including intermediate solutions in plan-then-merge), but inclusive of communication times in distributed local planning.

Where all runs of a particular experiment failed, the figures are replaced with a code to indicate the type of failure:

- *t/out (p)* indicates time out during planning;
- *t/out (m)* indicates time out during plan merging;
- *fail (p)* indicates failure during planning (all possibilities were exhausted);
- *fail (m)* indicates failure out during plan merging.

Entries are left blank where an experiment was not run, usually because there were fewer agents than level 1 tasks to distribute.

More detailed versions of the results in this section, including numbers of iterations of each algorithm, plan merging times in plan-then-merge and numbers of resets in distributed local planning, are provided in Appendix C.

5.4.1 Blocksworld problems

bwSwap problems These problems demonstrate a situation in which plan-then-merge and distributed local planning generally perform better than centralised planning. For problems of a significant size, the time saving should increase as more agents are added to the problem. However, the general performance of the planning mechanism prevented the investigation of enough problems to provide conclusive results about this trend (see below).

Table 5.2 shows success rates and solution times for *bwSwap_2* to *bwSwap_10*. Plan-then-merge and distributed local planning do better than centralised planning on all problems, with plan-then-merge being the fastest approach.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwSwap_2</i>	100% 420						
<i>bwSwap_4</i>	100% 13574	100% 5096			100% 8874		
<i>bwSwap_6</i>	100% 391600	80% 41898	100% 9211		100% 194844	100% 64870	
<i>bwSwap_8</i>	t/out (p)	80% 99337	100% 98714	100% 15698	t/out (p)	t/out (p)	t/out (p)
<i>bwSwap_10</i>	t/out (p)	t/out (p)	60% 217571	40% 223002	t/out (p)	t/out (p)	t/out (p)

Table 5.2: Success rates and average solution times (in milliseconds) for *bwSwap* problems.

The numbers of tasks in successfully generated solutions are shown in Table 5.3. These figures act as a rough guide to the quality of the plans produced. While the planning algorithms do not explicitly search for optimal plans, all algorithms tend to favour shorter plans, as plans with fewer tasks generally involve fewer potential conflicts. Plan-then-merge and distributed local planning may be at a disadvantage in some problems, however, because they divide search up into smaller components. In this case, all algorithms produced optimal plans.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwSwap_2</i>	2						
<i>bwSwap_4</i>	4	4			4		
<i>bwSwap_6</i>	6	6	6		6	6	
<i>bwSwap_8</i>		8	8	8			
<i>bwSwap_10</i>			10	10			

Table 5.3: Average numbers of tasks per plan for *bwSwap* problems.

bwReverse problems These problems demonstrate the importance of the assignment of tasks in the decentralised algorithms. A problem *bwReverse_x* consists of $x - 1$ *achieve_on* tasks. Each task has an identical initial task tree to all the others, both in terms of the number of nodes

in the tree and the number of potential conflicts with other tasks. However, the distribution of tasks does cause the performance of plan-then-merge and distributed local planning to vary greatly.

The cost of plan merging increases dramatically with respect to the number of redundant tasks produced during the planning stage. This happens as the size of the problem and number of agents increase, as can be seen for two and three agent plan-then-merge applied to *bwReverseSeq* problems. Figure 5.5 shows that, while planning times for three agents are lower than those for two agents, increased plan merging times more than remove this advantage.

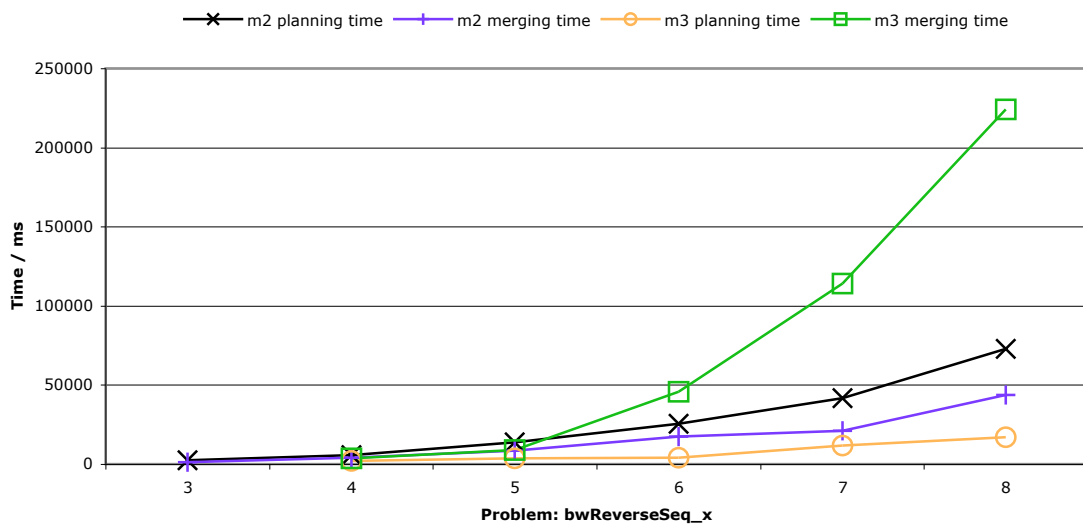


Figure 5.5: Planning and plan merging times for two agent and three agent plan-then-merge for *bwReverseSeq* problems.

The number of redundant tasks produced in *bwReverseSeq* problems is also higher than for *bwReverseRobin* problems, as illustrated by the following example:

Consider the *bwReverse_10* problem being solved by three agents: ignoring task ownership, the initial joint task tree is the same regardless of task distribution. However, the individual plans in Figure 5.6, generated during three agent plan-then-merge applied to *bwReverseRobin_10* and *bwReverseSeq_10*, are of different lengths. The *bwReverseRobin* plans are a total of six tasks shorter than their *bwReverseSeq* equivalents. This makes the plan merging step much simpler as

there are few possible task orderings and choices of *noop* decomposition to consider.

<i>bwReverseRobin</i>			<i>bwReverseSeq</i>		
<i>planner₁</i>	<i>planner₂</i>	<i>planner₃</i>	<i>planner₁</i>	<i>planner₂</i>	<i>planner₃</i>
<i>A</i> → <i>Table</i>	<i>A</i> → <i>Table</i>	<i>A</i> → <i>Table</i>	<i>A</i> → <i>Table</i>	<i>A</i> → <i>Table</i>	<i>A</i> → <i>Table</i>
<i>B</i> → <i>A</i>	<i>B</i> → <i>Table</i>	<i>B</i> → <i>Table</i>	<i>B</i> → <i>A</i>	<i>B</i> → <i>Table</i>	<i>B</i> → <i>Table</i>
<i>C</i> → <i>B</i>	<i>C</i> → <i>Table</i>	<i>C</i> → <i>Table</i>	<i>C</i> → <i>Table</i>	<i>C</i> → <i>B</i>	<i>C</i> → <i>Table</i>
<i>D</i> → <i>C</i>	<i>D</i> → <i>Table</i>	<i>D</i> → <i>Table</i>	<i>D</i> → <i>Table</i>	<i>D</i> → <i>Table</i>	<i>D</i> → <i>D</i>
	<i>E</i> → <i>D</i>	<i>E</i> → <i>Table</i>	<i>E</i> → <i>D</i>	<i>E</i> → <i>Table</i>	<i>E</i> → <i>Table</i>
	<i>F</i> → <i>E</i>	<i>F</i> → <i>Table</i>	<i>F</i> → <i>Table</i>	<i>F</i> → <i>E</i>	<i>F</i> → <i>Table</i>
	<i>G</i> → <i>F</i>	<i>G</i> → <i>Table</i>	<i>G</i> → <i>Table</i>	<i>G</i> → <i>Table</i>	<i>G</i> → <i>F</i>
		<i>H</i> → <i>G</i>	<i>H</i> → <i>G</i>	<i>H</i> → <i>Table</i>	<i>H</i> → <i>Table</i>
		<i>I</i> → <i>H</i>		<i>I</i> → <i>H</i>	<i>I</i> → <i>Table</i>
		<i>J</i> → <i>I</i>			<i>J</i> → <i>I</i>

Figure 5.6: Individual plans produced during three agent plan-then-merge for *bwReverseRobin₁₀* and *bwReverseSeq₁₀*.

Interestingly, the planning times in Table 5.4 show that plan-then-merge actually does much worse on *bwReverseRobin* problems than *bwReverseSeq* problems. Performance on *bwReverseRobin* is also routinely much worse than that of centralised planning, with many experiments timing out on every run. The timeouts occur during planning rather than merging, which indicates that other factors are affecting performance. It is hypothesised that when adjacent pairs of blocks are considered within the same plan, the redundancy in *achieve_clear* tasks causes the planner to favour them for refinement over other tasks. This is a sensible policy as it sets up causal links for achieving multiple *achieve_on* tasks. The round robin task distribution prevents this preference forming, making planning harder for fewer tasks.

Distributed local planning is also affected by task distribution, although in a different manner than plan-then-merge. Agents are able to remove duplicate tasks as they are discovered, keeping redundancy lower during planning. If anything, distributed local planning benefits from round robin task assignment. The agents share duplicate *achieve_clear* tasks between them, and can detect the corresponding increase in the number of conflicts through the shared external summary information. Distributed local planning never does as well as rival approaches, but is less severely affected by task distribution than plan-then-merge.

5.4. Comparison of planning approaches

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwReverseRobin_2</i>	100% 441						
<i>bwReverseRobin_3</i>	100% 2295	100% 5114			100% 4054		
<i>bwReverseRobin_4</i>	100% 6360	100% 15100	t/out (m)		100% 13244	100% 64630	
<i>bwReverseRobin_5</i>	100% 16812	40% 131242	80% 60928	t/out (m)	100% 33419	100% 49943	t/out (p)
<i>bwReverseRobin_6</i>	100% 45553	t/out (p)	t/out (p)	40% 471473	100% 84071	100% 116066	20% 258353
<i>bwReverseRobin_7</i>	100% 129446	t/out (p)	t/out (p)	t/out (m)	100% 228933	t/out (p)	t/out (p)
<i>bwReverseRobin_8</i>	100% 339387	t/out (p)	t/out (p)	t/out (p)	100% 495945	t/out (p)	t/out (p)
<i>bwReverseRobin_9</i>	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseRobin_10</i>	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_2</i>	100% 441						
<i>bwReverseSeq_3</i>	100% 2295	40% 3525			100% 7453		
<i>bwReverseSeq_4</i>	100% 6360	40% 9641	40% 9008		100% 40472	100% 60339	
<i>bwReverseSeq_5</i>	100% 16812	40% 22153	40% 21583	t/out (m)	100% 102875	100% 475733	40% 387466
<i>bwReverseSeq_6</i>	100% 45553	40% 42681	40% 95460	t/out (m)	100% 452123	40% 624514	t/out (p)
<i>bwReverseSeq_7</i>	100% 129446	40% 62880	40% 239623	t/out (m)	80% 590276	t/out (p)	t/out (p)
<i>bwReverseSeq_8</i>	100% 339387	40% 116367	40% 465592	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_9</i>	t/out (p)	t/out (m)	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_10</i>	t/out (p)	t/out (m)	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)

Table 5.4: Success rates and average solution times (in milliseconds) for *bwReverse* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwReverseRobin_2</i>	2						
<i>bwReverseRobin_3</i>	3	3			3.2		
<i>bwReverseRobin_4</i>	4	5			4	5.4	
<i>bwReverseRobin_5</i>	5	7	6		5	5	
<i>bwReverseRobin_6</i>	6			7	6	6	6
<i>bwReverseRobin_7</i>	7				7		
<i>bwReverseRobin_8</i>	8				8		
<i>bwReverseRobin_9</i>							
<i>bwReverseRobin_10</i>							
<i>bwReverseSeq_2</i>	2						
<i>bwReverseSeq_3</i>	3	3			3.2		
<i>bwReverseSeq_4</i>	4	4	4		5	4.6	
<i>bwReverseSeq_5</i>	5	5	5		5	6	6
<i>bwReverseSeq_6</i>	6	6	6		7	8	
<i>bwReverseSeq_7</i>	7	7	7		7		
<i>bwReverseSeq_8</i>	8	8	8				
<i>bwReverseSeq_9</i>							
<i>bwReverseSeq_10</i>							

Table 5.5: Average numbers of tasks per plan for *bwReverse* problems.

The numbers of tasks in successfully generated plans are shown in Table 5.5. Centralised planning consistently produces optimal plans, and plan-then-merge and distributed local planning typically produce plans that are optimal or near optimal³. From the small amount of evidence available (plan lengths for *bwReverseRobin_5* and *bwReverseRobin_6*) it is hypothesised that, if plan-then-merge had been more successful on *bwReverseRobin* problems, it would have produced lower quality solutions than distributed local planning. However, this cannot be confirmed with the few figures available.

bwRandom problems The results for *bwRandom* problems are shown in Table 5.6 and graphically in Figure 5.7. The fastest algorithm is normally either centralised planning or plan-then-merge. The results for plan-then-merge show increasing figures for large numbers of agents in some problems, and decreasing figures in others. This indicates that some problems decompose well into separate individual problems and some do not. Distributed local planning, while never being quite as fast as centralised planning, is slightly more consistent than plan-then-merge, successfully solving many problems that otherwise time out during merging.

Problem	c		m2		m3		m4		d2		d3		d4	
<i>bwRandom_4_1</i>	100%	8360	100%	18215	100%	14445	100%	44532	100%	31361	100%	72749	100%	92423
<i>bwRandom_4_2</i>	100%	2558	100%	3673	100%	3026	100%	3726	100%	4125	100%	5618	100%	11889
<i>bwRandom_4_3</i>	100%	118833	t/out (p)		100%	9764	t/out (m)		t/out (p) *		t/out (p) *		t/out (p) *	
<i>bwRandom_4_4</i>	100%	4480	100%	6345	100%	6375	100%	7122	100%	11430	100%	11358	100%	19632
<i>bwRandom_4_5</i>	100%	4906	100%	12421	100%	42636	t/out (m)		100%	18766	100%	31459	100%	43032
<i>bwRandom_4_6</i>	100%	297896	100%	57232	100%	58685	100%	59680	80%	391353	100%	160838	100%	234950
<i>bwRandom_4_7</i>	100%	6512	100%	12293	100%	15020	100%	14532	100%	40222	100%	38672	100%	86273
<i>bwRandom_4_8</i>	100%	7286	100%	14364	100%	3763	100%	3771	100%	19651	100%	9010	100%	11651
<i>bwRandom_4_9</i>	100%	6963	100%	20981	100%	34694	100%	35026	100%	69792	100%	44140	100%	87729
<i>bwRandom_4_10</i>	100%	123348	100%	199444	t/out (m)		t/out (m)		t/out (p)		100%	76188	100%	99904
<i>bwRandom_4_11</i>	100%	7659	100%	15521	100%	25038	t/out (m)		100%	34721	100%	49821	100%	124590
<i>bwRandom_4_12</i>	100%	2914	100%	4564	100%	8190	100%	4734	100%	6909	100%	11372	100%	17636
<i>bwRandom_4_13</i>	100%	10407	100%	14512	100%	14641	100%	21348	100%	39633	100%	50072	80%	228519
<i>bwRandom_4_14</i>	t/out (p)		t/out (m)		100%	35743	t/out (m)		100%	160094	100%	126994	100%	174400
<i>bwRandom_4_15</i>	100%	7527	100%	27180	100%	17067	t/out (m)		100%	18043	100%	45318	100%	121318
<i>bwRandom_4_16</i>	100%	3729	100%	9005	100%	6152	100%	6008	100%	9212	100%	15796	100%	23795
<i>bwRandom_4_17</i>	100%	33131	t/out (m)		t/out (m)		t/out (m)		t/out (p)		100%	314004	100%	417068
<i>bwRandom_4_18</i>	100%	52626	100%	7930	t/out (m)		t/out (m)		t/out (p) *		100%	69144	t/out (p) *	
<i>bwRandom_4_19</i>	t/out (p)		t/out (p)		t/out (p)		t/out (m)		t/out (p)		t/out (p) *		t/out (p) *	
<i>bwRandom_4_20</i>	100%	154815	100%	9142	t/out (m)		t/out (m)		t/out (p) *		t/out (p) *		t/out (p) *	

Table 5.6: Success rates and average solution times (in milliseconds) for *bwRandom* problems. * indicates that one or more runs failed due to weight explosion.

The number of tasks in solution plans is shown in Table 5.7. Many problems show the same patterns as before: centralised planning produced the shortest plans while plan-then-merge and

³Sub-optimality arises from a tendency to move blocks via the table when they could be moved directly.

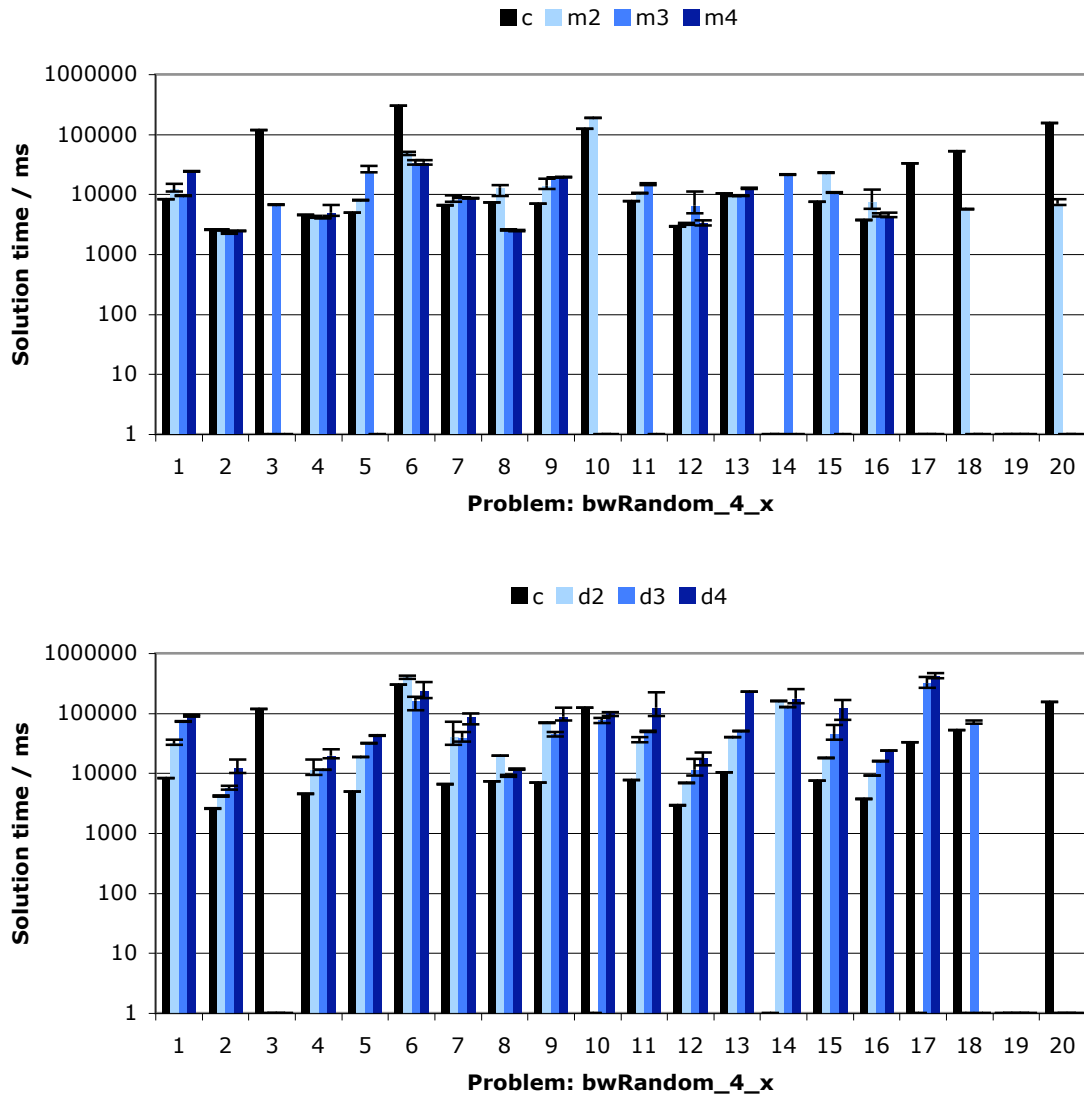


Figure 5.7: Average solution times for *bwRandom* problems. Error bars represent standard error calculations based on the average, maximum and minimum times recorded for each experiment.

distributed local planning produce plans that are of equal length or longer. However, in some problems (*bwRandom_4_3*, *_7*, *_9*, *_18* and *_20*) plan-then-merge and distributed local planning actually outperform centralised planning in this respect.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwRandom_4_1</i>	4	4	4	4	4.2	5	4.8
<i>bwRandom_4_2</i>	1	1	1	1	1	1	1
<i>bwRandom_4_3</i>	5		4				
<i>bwRandom_4_4</i>	2	2	2	2	2	2	2
<i>bwRandom_4_5</i>	4	4	4		4	4	4
<i>bwRandom_4_6</i>	5	5	5	5	6	6	6
<i>bwRandom_4_7</i>	4	3	3	3	3.8	3.6	3.2
<i>bwRandom_4_8</i>	2	2	2	2	2	2	2
<i>bwRandom_4_9</i>	5	4	4	4	5	4.6	5
<i>bwRandom_4_10</i>	6	6				6	6
<i>bwRandom_4_11</i>	4	4	4		4	4.6	4.2
<i>bwRandom_4_12</i>	2	2	2	2	2	2	2
<i>bwRandom_4_13</i>	4	4	4	4	4	4	4.6
<i>bwRandom_4_14</i>			6		6	6	6
<i>bwRandom_4_15</i>	3	3	3		3	3.6	3.6
<i>bwRandom_4_16</i>	2	2	2	2	2	2	2
<i>bwRandom_4_17</i>	6					6	6
<i>bwRandom_4_18</i>	4	3				4	
<i>bwRandom_4_19</i>							
<i>bwRandom_4_20</i>	5	4					

Table 5.7: Average numbers of tasks per plan for *bwRandom* problems.

Weight explosions in distributed local planning The results for *bwRandom* problems show the first of several bits of evidence of erratic behaviour in distributed local planning, which occasionally exhibited a pattern of failure that will be referred to as *weight explosion*. Relevant experiments are marked with an asterisk in Table 5.6.

In this situation the agents get caught in a loop which is not broken by increasing summary weights. A plot of heuristic value against time for an example run is shown in Figure 5.8. A cycle is visible where the agents search a set of partial plans before resetting, increasing their summary weights and repeating the process. After the third repetition shown the weights overflow: the agents get stuck at high weight values for the remainder of the 10 minute time limit.

Weight explosion occurs consistently in *bwRandom* problems: if a particular algorithm fails due to weight explosion on one run, it fails again on all the other runs of the same problem. The phenomenon also occurs in some *holesSpecial* problems (Section 5.4.3), although runs of these problems are a mixture of explosions and successful solutions. It appears that weight explosion

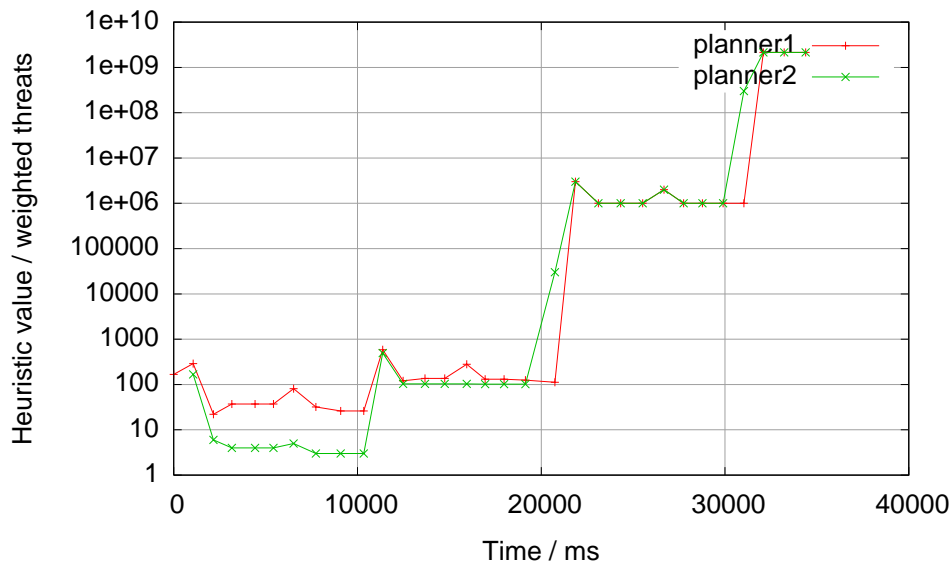


Figure 5.8: Plot of heuristic value against time for three agent distributed local planning on *bwRandom_4_3* showing weight explosion in distributed local planning. The x axis is truncated: both plots actually extend horizontally all the way to the 10 minute time limit.

is a particular type of situation in which agents get trapped in a local heuristic minimum, and the summary weights are insufficient to help them escape. With a proper implementation of *nogood* constraints in which agents are prevented from revisiting sets of plans that have caused a reset, it would be impossible for this cycle to happen indefinitely.

5.4.2 Navigation problems

Navigation problems are fundamentally different to Blocksworld problems in that each agent is only allowed to control a specific subset of the resources (in this case robots) in the environment. This highlights a key weakness in the plan-then-merge approach. As mentioned in Section 4.3, planning fails if one or more planner agents is unable to create a valid individual plan. However, in this domain agents are reliant on one another to move robots out of the way so their robots can reach their destinations.

Initial conditions and plan-then-merge Without special treatment plan-then-merge fails to solve all but the most trivial problems in this domain. Each planning agent receives an initial task tree containing the *travel* tasks for its robots and the initial conditions for all robots. The

agent is only able to move its own robots, and it is unaware during planning that the other robots are being moved by other agents. Consider the multi robot *navLine* and *navRing* problems in Figures 5.2 and 5.4: in each case there is at least one robot which has to pass the start point of another robot to reach its destination.

To work around this problem, agents in plan-then-merge are only informed of the initial positions of their own robots. By restricting access to this knowledge many more problems are made solvable. *All Navigation experiments involving plan-then-merge were modified in this way.* Distributed local planning is unaffected by this: agents are able to share information about their plans as they are built up, so they are aware of the movement of each others' robots and are able to indirectly request changes in routes by issuing *reset* commands.

navLine problems The planning times for *navLine* problems are shown in Table 5.8. Again, centralised planning and plan-then-merge have similar performance. Figure 5.9 shows the relative durations of the planning and merging phases of plan-then-merge and the total durations of centralised planning and plan-then-merge. Two trends are noticeable. Firstly, the merging phase of plan-then-merge occupies more time as the number of agents increases. Secondly, plan-then-merge tends to perform better on larger problems involving more robots and more locations: on smaller problems the cost of the merging phase is prohibitively high.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navLine_3_1</i>	100% 579						
<i>navLine_3_2</i>	100% 579	100% 1220			100% 1114		
<i>navLine_4_1</i>	100% 1402						
<i>navLine_4_2</i>	100% 2067	100% 3271			100% 2717		
<i>navLine_4_3</i>	100% 1068	100% 1899	100% 2087		100% 1983	100% 2242	
<i>navLine_5_1</i>	100% 2559						
<i>navLine_5_2</i>	100% 5528	100% 10004			100% 98392		
<i>navLine_5_3</i>	100% 5717	100% 5635	100% 7965		100% 8197	100% 12333	
<i>navLine_5_4</i>	100% 1728	100% 3265	100% 2905	100% 3436	100% 2939	100% 5837	40% 8999
<i>navLine_6_1</i>	100% 5235						
<i>navLine_6_2</i>	100% 24512	100% 25850			100% 283050		
<i>navLine_6_3</i>	100% 23382	100% 23294	100% 26416		100% 30121	t/out (p)	
<i>navLine_6_4</i>	100% 14803	100% 14316	100% 12633	100% 16434	20% 28385	100% 68449	100% 87357
<i>navLine_6_5</i>	100% 3413	100% 5145	100% 5063	100% 25608	100% 7409	100% 14262	100% 15886

Table 5.8: Success rates and average solution times (in milliseconds) for *navLine* problems.

Average lengths of plans are shown in Table 5.9. Algorithms typically perform equally well in this respect, although all algorithms tend to produce Navigation plans with extra redundant tasks in them, as will be demonstrated below.

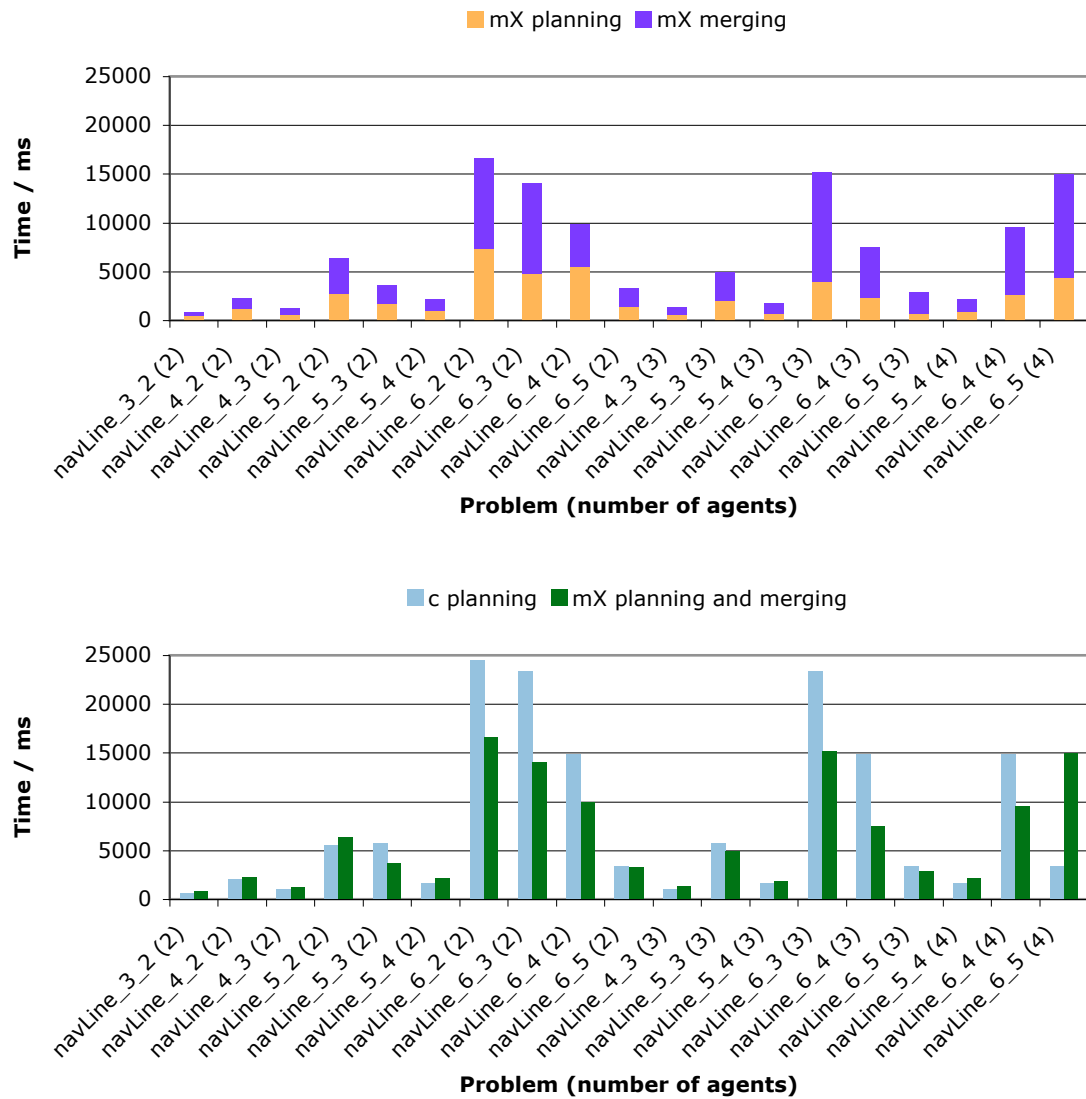


Figure 5.9: Comparison of solutions times for centralised planning and plan-then-merge for *navLine* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navLine_2_1</i>	1						
<i>navLine_3_1</i>	2						
<i>navLine_3_2</i>	2	2			2		
<i>navLine_4_1</i>	3						
<i>navLine_4_2</i>	4	4			4		
<i>navLine_4_3</i>	3	3	3		3	3	
<i>navLine_5_1</i>	4						
<i>navLine_5_2</i>	6	6			6		
<i>navLine_5_3</i>	6	6	6		6	6	
<i>navLine_5_4</i>	4	4	4	4	4	4	4
<i>navLine_6_1</i>	5						
<i>navLine_6_2</i>	10	8			8		
<i>navLine_6_3</i>	9	9	9		9		
<i>navLine_6_4</i>	8	8	8	8	8	8	8
<i>navLine_6_5</i>	5	5	5	5	5	5	5

Table 5.9: Average numbers of tasks per plan for *navLine* problems.

navRing problems The solution times for *navRing* problems are shown in Table 5.10. What is immediately apparent is that despite the alterations made to the initial conditions, plan-then-merge fails to solve almost all of the problems. What is more, the failures are all in the plan merging stage.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navRing_3_2</i>	100% 966	fail (m)			100% 2105		
<i>navRing_4_1</i>	100% 987						
<i>navRing_4_2</i>	100% 2103	fail (m)			100% 28274		
<i>navRing_4_3</i>	100% 9640	100% 7011	fail (m)		t/out (p)	100% 133506	
<i>navRing_5_1</i>	100% 1332						
<i>navRing_5_2</i>	100% 3855	fail (m)			100% 7605		
<i>navRing_5_3</i>	100% 197350	fail (m)	fail (m)		t/out (p)	100% 39137	
<i>navRing_5_4</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	t/out (p)
<i>navRing_6_1</i>	100% 3009						
<i>navRing_6_2</i>	100% 9950	fail (m)			100% 9884		
<i>navRing_6_3</i>	100% 453511	fail (m)			t/out (p)	100% 433342	
<i>navRing_6_4</i>	t/out (p)	t/out (m)	t/out (m)		t/out (p)	t/out (p)	t/out (p)
<i>navRing_6_5</i>	t/out (p)	t/out (p)	t/out (m)		t/out (p)	t/out (p)	t/out (p)

Table 5.10: Success rates and average solution times (in milliseconds) for *navRing* problems.

A hypothesis for this failure can be formed by looking at a typical successful plan from the centralised planning algorithm. The plan for *navRing_5_3* in Figure 5.10 shows that the agent has chosen a sub-optimal route for robot r_2 : the robot travels backward at one point in its journey. None of the planning algorithms from Chapter 4 search for optimal plans: each algorithm simply chooses the first successful plan it finds. If a planner is presented with two candidate plans of equal heuristic value, it will arbitrarily choose one plan based on the order in which they are formed. This can easily result in plans involving one or more sub-optimal backward steps, or in a single robot case, a single robot plan that travels the opposite way around the ring to the

others. In the example the joint plan is coordinated because the search process is centralised, but the tendency for “backward” travel is revealed. Given the likelihood of this behaviour appearing in individual plans, it is extremely likely that a set of non-mergeable individual plans is produced. The single successful result for *navRing_4_3* shows that it is possible (though unlikely) that the individual plans will be mergeable and a joint solution possible.

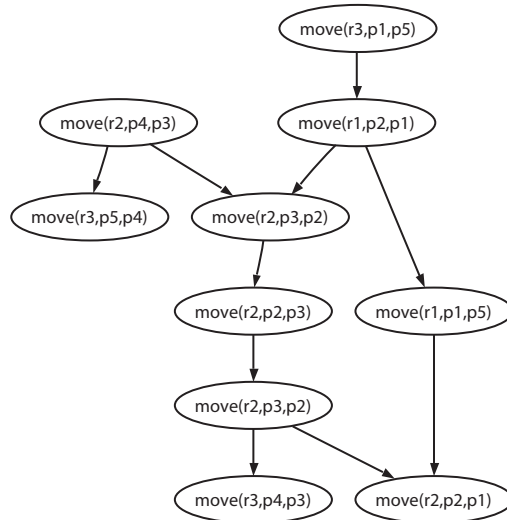


Figure 5.10: Plan produced by centralised planning for *navRing_5_3*, showing a sub-optimal route for robot r_2 around locations p_2 and p_3 .

Distributed local planning performs well when the number of robots matches the number of agents, and poorly when one agent is in control of several robots. A hypothesis for this follows:

Distributed local planning is effectively a blend of refinement based depth first search (DFS; Section 4.2) and asynchronous-backtracking-like removal of inter-plan threats (ABT; Section 4.4.1). When either the ABT or the DFS part of the algorithm dominates, the algorithm as a whole tends to do well. If DFS dominates, for example, the agents do not have to perform many resets and a solution is typically found very quickly. Similarly, if ABT dominates, resets are frequent but the recovery time from each is small, allowing the agents to quickly converge on a solution. When agents have to perform lots of resets and the time to recover from each reset is large, the time to solve the problem explodes in polynomial fashion as the product $number_resets \times average_recovery_time$ becomes large. In larger Navigation and Holes problems (Section 5.4.3), the large time taken per refinement, coupled with this explosion, quickly

causes the total solution time to rise over the 10 minute limit.

Numbers of tasks in solution plans are shown in Table 5.11. Unfortunately, success rates are too low for patterns or trends to be seen.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navRing_3_1</i>	1						
<i>navRing_3_2</i>	3				3		
<i>navRing_4_1</i>	2						
<i>navRing_4_2</i>	4				8		
<i>navRing_4_3</i>	6	6			6	7.2	
<i>navRing_5_1</i>	2						
<i>navRing_5_2</i>	5				5		
<i>navRing_5_3</i>	10					7	
<i>navRing_5_4</i>							
<i>navRing_6_1</i>	3						
<i>navRing_6_2</i>	6				6		
<i>navRing_6_3</i>	13					11	
<i>navRing_6_4</i>							
<i>navRing_6_5</i>							

Table 5.11: Average numbers of tasks per plan for *navRing* problems.

navStar problems The solution times for *navStar* problems are shown in Table 5.12. Given the performance of plan-then-merge on *navRing* problems, it is unsurprising that it fails during merging on many *navStar* problems, and particularly on problems with a high ratio of robots to locations. Distributed local planning fails to solve many problems within the time and memory limits, for the same reasons described above and the reasons below.

Plateaux in distributed local planning A pattern can be seen among many of the failed runs of distributed local planning on Navigation problems. Quite often, runs seem to fail because the heuristic weights of the agents stop changing, forming a “plateau” of heuristic value. An example of this is shown in Figure 5.11. This graph is typical of many of the failed runs from *navLine*, *navRing* and *navStar* problems, and may suggest a second type of situation in which distributed local planning gets stuck in a dead end from which it cannot escape.

There is evidence in some cases to suggest that distributed local planning does not get permanently caught in plateaux, but merely spends a long time in them. For example, Figure 5.12 shows another run from the same experiment shown in Figure 5.11. In this figure the planners clearly escape from the plateau and find a solution.

5.4. Comparison of planning approaches

Problem	c	m2	m3	m4	d2	d3	d4
navStar_3_2_2	100% 367634	fail (m)			80% 182565		
navStar_3_2_3	100% 81359	fail (m)			t/out (p)		
navStar_3_2_4	100% 160218	fail (m)			20% 53984		
navStar_3_2_5	100% 15090	100% 10614			100% 66902		
navStar_3_2_6	100% 7072	100% 9841			100% 66697		
navStar_3_2_7	100% 8846	100% 8222			100% 13076		
navStar_3_2_8	100% 233345	fail (m)			t/out (p)		
navStar_3_2_9	100% 22882	fail (m)			100% 17354		
navStar_3_2_10	100% 177119	fail (m)			100% 82791		
navStar_4_2_1	100% 25190	100% 18399			100% 261841		
navStar_4_2_2	100% 9295	100% 10308			100% 31836		
navStar_4_2_3	80% 487028	fail (m)			80% 537830		
navStar_4_2_4	t/out (p)	fail (m)			80% 359805		
navStar_4_2_5	100% 33235	100% 21851			100% 390803		
navStar_4_2_6	100% 43377	100% 27605			t/out (p)		
navStar_4_2_7	t/out (p)	fail (m)			t/out (p)		
navStar_4_2_8	t/out (p)	fail (m)			80% 298879		
navStar_4_2_9	t/out (p)	fail (m)			t/out (p)		
navStar_4_2_10	t/out (p)	fail (m)			t/out (p)		
navStar_4_3_1	100% 11395	100% 42282	100% 21283		100% 36974	100% 64621	
navStar_4_3_2	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_4_3_3	100% 19770	100% 15395	100% 21584		100% 53078	80% 96707	
navStar_4_3_4	100% 386381	100% 239789	fail (m)		t/out (p)	t/out (p)	
navStar_4_3_5	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_4_3_6	t/out (p)	fail (p)	fail (m)		80% 527918	20% 564518	
navStar_4_3_7	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_4_3_8	100% 48835	100% 44555	100% 19228		100% 270059	20% 198028	
navStar_4_3_9	100% 170716	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_4_3_10	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
navStar_5_2_1	100% 302786	fail (m)			t/out (p)		
navStar_5_2_2	100% 26801	100% 33942			t/out (p)		
navStar_5_2_3	t/out (p)	fail (m)			t/out (p)		
navStar_5_2_4	t/out (p)	fail (m)			t/out (p)		
navStar_5_2_5	100% 366400	fail (m)			t/out (p)		
navStar_5_2_6	t/out (p)	fail (m)			t/out (p)		
navStar_5_2_7	100% 26419	100% 35008			t/out (p)		
navStar_5_2_8	100% 84023	100% 46943			t/out (p)		
navStar_5_2_9	100% 85030	100% 49336			t/out (p)		
navStar_5_2_10	100% 247315	100% 40543			t/out (p)		
navStar_5_3_1	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_5_3_2	100% 95264	100% 82921	100% 56255		t/out (p)	20% 304508	
navStar_5_3_3	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
navStar_5_3_4	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
navStar_5_3_5	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
navStar_5_3_6	80% 365514	100% 399564	100% 57791		t/out (p)	t/out (p)	
navStar_5_3_7	100% 132781	100% 68507	100% 63369		t/out (p)	t/out (p)	
navStar_5_3_8	100% 232181	100% 43207	100% 60782		t/out (p)	t/out (p)	
navStar_5_3_9	100% 278276	fail (m)	100% 76573		100% 187274	t/out (p)	
navStar_5_3_10	100% 417474	fail (p)	100% 72574		t/out (p)	t/out (p)	
navStar_5_4_1	t/out (p)	fail (m)	t/out (p)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_2	t/out (p)	fail (m)	fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_3	t/out (p)	fail (m)	fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_4	t/out (p)	fail (m)	fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_5	t/out (p)	fail (m)	t/out (p)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_6	t/out (p)	fail (m)	fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_7	t/out (p)	t/out (p)	t/out (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_8	t/out (p)	t/out (p)	fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_9	t/out (p)	fail (m)	t/out (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
navStar_5_4_10	t/out (p)	t/out (p)	t/out (p)	fail (m)	t/out (p)	t/out (p)	t/out (p)

Table 5.12: Success rates and average planning times (in milliseconds) for *navStar* problems.

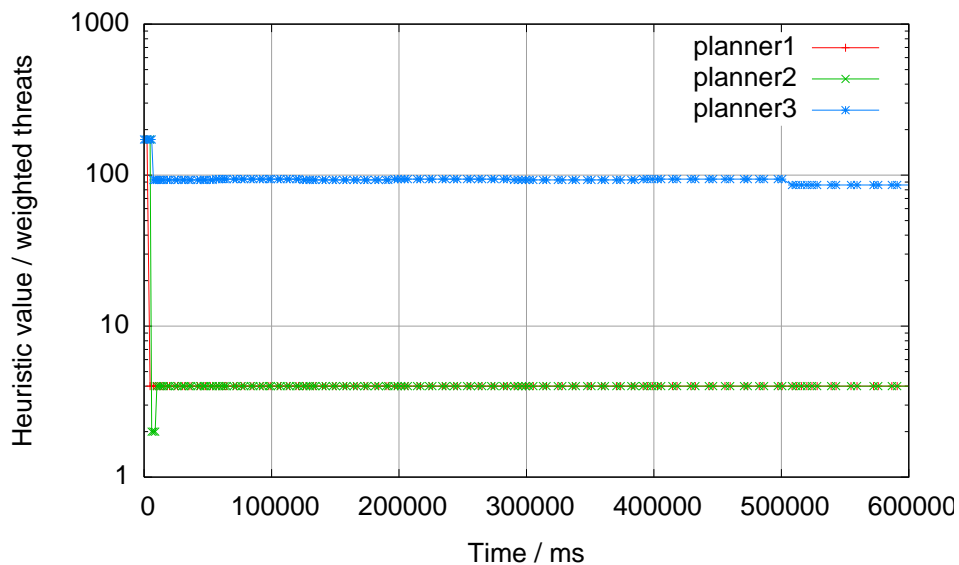


Figure 5.11: Plot of heuristic value against time for three agent distributed local planning on *navStar_4_3_8* showing a heuristic plateau in distributed local planning.

Plateaux are caused by agents visiting large numbers of plans with the same heuristic value. These plans generally contain different groundings or different orderings of the same set of tasks. In distributed local planning, the effect is worse as agents often have to wait for one another to create concrete effectors before they can investigate possible new plans (Section 4.4.2). This can cause a succession of plateaux from which it takes a very long time to escape.

The situation is made worse still by the lack of a check for equality or dominance when adding plans to the open list. An *equality* check determines whether a newly produced plan is the same as another plan that is already *open* (on the open list) or *closed* (already visited). A *dominance* check determines whether there is a currently open or closed plan that has the same effects as a new plan but is less costly to perform (takes less time or resources). A dominance test was included in CHiPs (Chapter 6 of Clement, 2002), but with the introduction of planning variables and the possibility of equivalent plans with alternative variable namings it was considered too costly for inclusion in MPF. The decompose-then-order refinement strategy in MPF is designed to minimise the production of redundant plans, but some plans with equivalent orderings are still produced.

Numbers of tasks in solution plans are shown in Table 5.13. Plan-then-merge tends to perform the best in the problems it can solve. A hypothesis for this lies in the ability of plan-then-merge

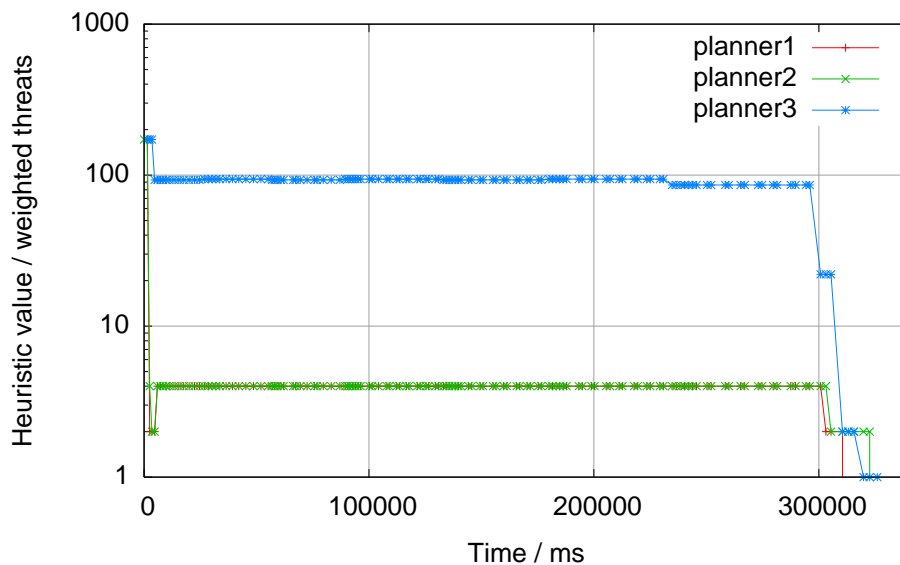


Figure 5.12: Plot of heuristic value against time for a different run of the experiment in Figure 5.11 showing escape from a heuristic plateau.

to directly remove tasks from the final joint plan. Centralised and distributed local planning have similar average performance.

5.4.3 Holes problems

Two types of problems were considered in the Holes domain: *holesGeneral* problems are similar to Navigation problems in that they involve agents with restricted control over resources in the environment, while *holesSpecial* problems impose further restrictions in terms of the knowledge the agents have of the environment.

MPF does not directly allow the domain designer to specify the knowledge of external state information on a per agent basis. However, limited knowledge can be simulated by limiting the types of preconditions of agents' tasks. This is the purpose of the *match_shape*, *match_color* and *match_size* methods and their counterparts, *verify_shape*, *verify_color* and *verify_size* (Section 5.2.3). Each method contains preconditions for just one of the three types of feature. By appropriately assigning methods of one type to each agent, the domain designer can restrict the type of state that the agent can reason about, effectively “blinding” it to the other types of feature. *holesGeneral* problems require agents to be able to reason about all types of feature,

Problem	c	m2	m3	m4	d2	d3	d4
navStar_3_2_1	9				9		
navStar_3_2_2	9				9		
navStar_3_2_3	9						
navStar_3_2_4	10				8		
navStar_3_2_5	7	5			7		
navStar_3_2_6	5	5			7		
navStar_3_2_7	5	5			5		
navStar_3_2_8	9						
navStar_3_2_9	9				5		
navStar_3_2_10	10				8		
navStar_4_2_1	6	6			12		
navStar_4_2_2	4	4			4		
navStar_4_2_3	11				9		
navStar_4_2_4					8		
navStar_4_2_5	6	6			6		
navStar_4_2_6	8	8					
navStar_4_2_7							
navStar_4_2_8					8		
navStar_4_2_9							
navStar_4_2_10							
navStar_4_3_1	5	5	5		5	5	
navStar_4_3_2							
navStar_4_3_3	6	6	6		6	6	
navStar_4_3_4	13	9					
navStar_4_3_5							
navStar_4_3_6					9	7	
navStar_4_3_7							
navStar_4_3_8	11	5	5		11	8	
navStar_4_3_9	12						
navStar_4_3_10							
navStar_5_2_1	13						
navStar_5_2_2	5	5					
navStar_5_2_3							
navStar_5_2_4							
navStar_5_2_5	11						
navStar_5_2_6							
navStar_5_2_7	5	5					
navStar_5_2_8	7	7					
navStar_5_2_9	8	8					
navStar_5_2_10	13	7					
navStar_5_3_1							
navStar_5_3_2	8	8	8			8	
navStar_5_3_3							
navStar_5_3_4							
navStar_5_3_5							
navStar_5_3_6	15	15	9				
navStar_5_3_7	8	8	8				
navStar_5_3_8	13	7	7				
navStar_5_3_9	14		8		8		
navStar_5_3_10	16		10				
navStar_5_4_1							
navStar_5_4_2							
navStar_5_4_3							
navStar_5_4_4							
navStar_5_4_5							
navStar_5_4_6							
navStar_5_4_7							
navStar_5_4_8							
navStar_5_4_9							
navStar_5_4_10							

Table 5.13: Average numbers of tasks per plan for *navStar* problems.

requiring a seventh method, *match_all*.

Domain specific optimisations Initial results for Holes problems (not presented here) showed extremely poor performance across all algorithms. This was surprising, as Holes is a non-recursive domain in which very large task trees cannot be created.

The reason for this poor performance was found to be related to MPF’s poor handling of “*static*” literals that cannot be changed by actions in the plan. The problem and solution are discussed below:

Consider the *holesGeneral* problem shown in Figure 5.13. During the task tree generation phase, the *simplify* function (Section 3.4.4) decomposes the *locate* tasks producing the initial task tree in Figure 5.14. Planning essentially involves finding a conflict free set of assignments for *?hole₁* to *?hole₃*.

The agent can use the *decompose* refinement to ground any of the tasks or state constraints that have variables in them. It typically starts by grounding a *place* task, in which case it creates new plans for each possible binding of the relevant *?hole* variable. The trouble is that the decomposition of the task does not have a direct effect on the values of the relevant *?shape*, *?color* or *?size* variables. Because all of the relevant conflicts are to do with features, the agent is unable to tell which task grounding is the “best” to work on. The agent effectively chooses a random set of hole variable bindings, and iterates through a number of combinations of feature variable bindings before it backtracks and tries another set of holes. This phenomenon produces poor performance during planning, and especially in distributed local planning, where a lot of *reset* messages may be necessary to identify the most constraining features of the problem.

A workaround for this problem, shown in Figure 5.15, involves altering the *simplify* function such that feature variables are grounded as soon as possible after the decomposition of the relevant *place* task. This was a domain specific optimisation that could be extended to a domain independent case: in general it should be treated as a refinement as there may be more than one way of resolving relevant threats. However, in the Holes domain, level 1 threats routinely arise

```

⟨root, null, {⟨task1, locate(peg1), {...}⟩, ⟨task2, locate(peg2), {...}⟩,
  ⟨task3, locate(peg3), {...}⟩}, {}, {
  ⟨post1, shape(peg1, square)⟩, ⟨post2, color(peg1, red)⟩, ⟨post3, size(peg1, small)⟩,
  ⟨post4, shape(peg2, circle)⟩, ⟨post5, color(peg2, blue)⟩, ⟨post6, size(peg2, large)⟩,
  ⟨post7, shape(peg3, square)⟩, ⟨post8, color(peg3, blue)⟩, ⟨post9, size(peg3, large)⟩,
  ⟨post10, shape(hole1, square)⟩, ⟨post11, color(hole1, red)⟩, ⟨post12, size(hole1, small)⟩,
  ⟨post13, shape(hole2, circle)⟩, ⟨post14, color(hole2, blue)⟩, ⟨post15, size(hole2, large)⟩,
  ⟨post16, shape(hole3, square)⟩, ⟨post17, color(hole3, blue)⟩, ⟨post18, size(hole3, large)⟩},
  {...}, {}⟩

```

Figure 5.13: Example *holesGeneral* problem for single agent placing three pegs. The features of each peg and hole are completely specified. Ordering constraints are omitted: postconditions occur simultaneously before any tasks begin.

```

⟨root, null, {⟨task4, place(peg1, ?hole1), {}⟩, ⟨task5, place(peg2, ?hole2), {}⟩,
  ⟨task6, place(peg3, ?hole3), {}⟩}, {
  ⟨pre1, shape(?hole1, ?shape1)⟩, ⟨pre2, color(?hole1, ?color1)⟩, ⟨pre3, size(?hole1, ?size1)⟩,
  ⟨pre4, shape(?hole2, ?shape2)⟩, ⟨pre5, color(?hole2, ?color2)⟩, ⟨pre6, size(?hole2, ?size2)⟩,
  ⟨pre7, shape(?hole3, ?shape3)⟩, ⟨pre8, color(?hole3, ?color3)⟩, ⟨pre9, size(?hole3, ?size3)⟩}, {
  ⟨post1, shape(peg1, square)⟩, ⟨post2, color(peg1, red)⟩, ⟨post3, size(peg1, small)⟩,
  ⟨post4, shape(peg2, circle)⟩, ⟨post5, color(peg2, blue)⟩, ⟨post6, size(peg2, large)⟩,
  ⟨post7, shape(peg3, square)⟩, ⟨post8, color(peg3, blue)⟩, ⟨post9, size(peg3, large)⟩,
  ⟨post10, shape(hole1, square)⟩, ⟨post11, color(hole1, red)⟩, ⟨post12, size(hole1, small)⟩,
  ⟨post13, shape(hole2, circle)⟩, ⟨post14, color(hole2, blue)⟩, ⟨post15, size(hole2, large)⟩,
  ⟨post16, shape(hole3, square)⟩, ⟨post17, color(hole3, blue)⟩, ⟨post18, size(hole3, large)⟩},
  {...}, {...}⟩

```

Figure 5.14: Initial task tree for the problem in Figure 5.13, shown after simplification. Ordering and binding constraints are omitted: postconditions occur simultaneously before any tasks of preconditions begin, each task is ordered after all preconditions involving the same planning variables, and planning variables are completely unbound.


```

1  function simplify_plan(plan) :
2      call simplify(plan)
3      for each level 1 threat may_clobber(con1, con2):
4          node1 ← node(con1)
5          node2 ← node(con2)
6          if node1 and node2 are state constraints:
7              if there is 1 variable in total between literal(node1) and literal(node2):
8                  add constraints to vars(plan) such that literal(node1) ≠ literal(node2)

```

Figure 5.15: Algorithm for domain specific task tree simplification in Holes problems. The function calls the original simplify function (line 2) and then performs extra optimisations on level 1 state constraints.

that can *only* be resolved by binding planning variables. It is safe to resolve a threat as an extra stage of simplification if there is only *one* way it can be removed:

- the threat cannot be resolved by adding ordering constraints to the plan;
- the threat cannot be resolved by decomposition (the existence of each summary condition is *must*);
- the threat is between level 1 summary conditions (so that the variables involved can easily be discovered);
- the literals of the relevant summary conditions share exactly 1 ungrounded planning variable (so that bindings of alternative variables do not need to be considered).

With this optimisation in place, agents are able to make appropriate variable bindings as tasks are grounded, resulting in a large gain in planning efficiency. For example, the problem in Figure 5.13 can be solved by an agent in 1 refinement using the optimised simplification technique, as opposed to 37 refinements without optimisation. *The new version of the simplify function was used in all Holes problems.*

holesGeneral problems The results for these problems are shown in Table 5.14 and graphically in Figure 5.16. The problems posed little difficulty for centralised planning, which typically solved them in a single refinement. Plan-then-merge also showed good performance, typically being one or two seconds slower than centralised planning. This success is surprising given the inability of plan-then-merge to cope with Navigation problems, although there

is a straightforward explanation for it. In Navigation problems, planning agents tend to create very specific plans that produce unresolvable conflicts during plan merging. In Holes problems, however, the *simplify_plan* function in Figure 5.15 helps agents produce *abstract solutions* (Section 3.2.2) in which *?hole* variables are partially grounded such that pegs can be placed in any appropriate hole.

Consider, for example, a two agent plan-then-merge problem in which:

- *agent*₁ is trying to locate a large blue peg, *peg*₁;
- *agent*₂ is trying to locate a small blue peg, *peg*₂;
- there are two holes: *hole*₁ only accepts small pegs and *hole*₂ only accepts blue pegs.

In this problem, *agent*₁ would produce a plan in which *peg*₁ is placed in *hole*₁ and *agent*₂ would produce a plan in which *peg*₂ could be placed in *either* hole. These plans are mergeable, as the merger can add binding constraints to *agent*₂'s plan to make sure it is placed in the free hole.

The small speed advantage of centralised planning over plan-then-merge in the results is attributed to the extra time taken for the plan merging agent to remove *noop* decompositions from the task tree. The generally poor performance of distributed local planning is attributed to a combination of the number of *resets* required and the overheads of communication and maintenance of summary information. This is discussed further below.

Problem	c		m3		d3		Problem	c		m3		d3	
<i>holesGeneral_3_1_3_z</i>	100	510	94	2507	82	58780	<i>holesGeneral_4_1_3_z</i>	100	1321	100	3331	72	949408
<i>holesGeneral_3_1_4_z</i>	100	610	100	2264	76	50067	<i>holesGeneral_4_1_4_z</i>	100	1701	100	5998	84	353861
<i>holesGeneral_3_2_3_z</i>	100	219	100	1767	100	95960	<i>holesGeneral_4_2_3_z</i>	100	464	100	2709	94	417945
<i>holesGeneral_3_2_4_z</i>	100	146	100	1820	100	72118	<i>holesGeneral_4_2_4_z</i>	100	267	94	2693	84	182335
<i>holesGeneral_3_3_3_z</i>	100	280	94	1795	100	15161	<i>holesGeneral_4_3_3_z</i>	100	384	100	2678	90	62592
<i>holesGeneral_3_3_4_z</i>	100	309	88	1888	100	15236	<i>holesGeneral_4_3_4_z</i>	100	479	68	2661	100	39094

Table 5.14: Success rates and average planning times (in milliseconds) for *holesGeneral* problems. Results are averaged over 100 runs of 10 randomly generated problems.

holesSpecial problems The results for these problems are shown in Table 5.15 and graphically in Figure 5.17. Remember that the configuration of features of pegs and holes in any problem *holesSpecial_w_x_y_z* is the same as it is in the equivalent problem *holesGeneral_w_x_y_z*.

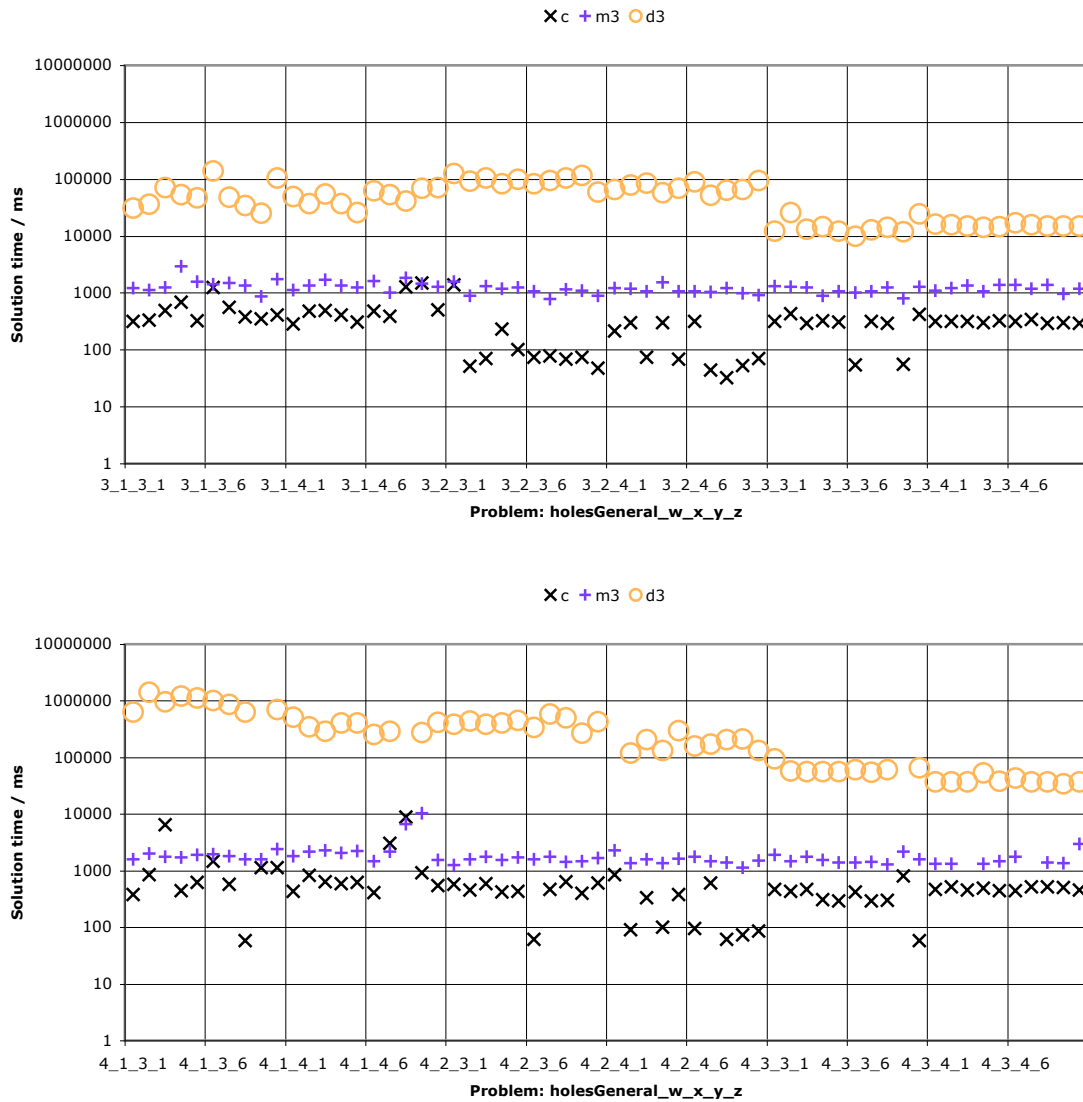


Figure 5.16: Average solution times for *holesGeneral* problems. Experiments are placed in groups of five for clarity. X axis labels refer to the problem name for the first experiment in each group: subsequent problems have successive *z* indices.

The times for centralised planning are much larger than their counterparts for *holesGeneral* problems, demonstrating that the structure of *holesSpecial* problems is much more convoluted: agents have three tasks to consider per peg and each task has two decompositions, as opposed to the single task and single decomposition in *holesGeneral* problems.

Plan-then-merge shows good performance relative to centralised planning, and even beats it in some cases (many *holesSpecial*_{3_2_4} problems, all *holesSpecial*_{4_1_4} and _{4_2_4} problems, and most *holesSpecial*_{4_3_4} problems), despite the fact that agents have access to different information. Again this is due to the agents' ability to find abstract solutions that can be merged into suitable plans afterwards. In other planning domains the individual plans produced will often not be so conveniently mergeable.

Distributed local planning times out on most of the problems here: this is unsurprising given that *holesGeneral*_{3_3_y_z} problems that took centralised planning up to 15 seconds to solve, take up to 5 minutes to in their equivalent *holesSpecial*_{3_3_y_z} forms. Again, this problem of speed is caused by a combination of the number of resets required and the extremely long time taken to perform refinements. The detailed results in Appendix C show the refinement times to be roughly equivalent to those in *holesGeneral* problems, but the numbers of refinements needed to be larger.

Holes problems are essentially constraint satisfaction problems. In their simplest form, they can be represented as a set of variables for peg/hole assignments, together with a set of constraints on which pegs can go where. MPF is a convoluted mechanism on which to base algorithms for solving these kinds of problems: far more complicated than equivalent CSP representations. There is much research in the CSP literature on these kinds of problems, including analytical and empirical analysis of *phase transitions* (Hogg et al., 1996; Rintanen, 2004), where the relative performance of different kinds of algorithms varies dramatically as the density and pattern of constraints changes. Further research with alternative planning mechanisms would be useful to find and analyse equivalent phase changes in multi agent planning.

5.4. Comparison of planning approaches

Problem	c	d3		Problem	c	d3	
<i>holesSpecial_3_1_3_z</i>	100	27337	8 199969	<i>holesSpecial_4_1_3_z</i>	80	129241	0 0
<i>holesSpecial_3_1_4_z</i>	100	33401	0 0	<i>holesSpecial_4_1_4_z</i>	100	186237	0 0
<i>holesSpecial_3_2_3_z</i>	100	38034	4 65119	<i>holesSpecial_4_2_3_z</i>	100	145646	0 0
<i>holesSpecial_3_2_4_z</i>	100	43133	8 113394	<i>holesSpecial_4_2_4_z</i>	100	170415	0 0
<i>holesSpecial_3_3_3_z</i>	100	20498	54 283510	<i>holesSpecial_4_3_3_z</i>	100	69642	0 0
<i>holesSpecial_3_3_4_z</i>	100	23807	70 273792	<i>holesSpecial_4_3_4_z</i>	100	90336	6 97891

Table 5.15: Success rates and average planning times (in milliseconds) for *holesSpecial* problems. Results are averaged over 100 runs of 10 randomly generated problems.

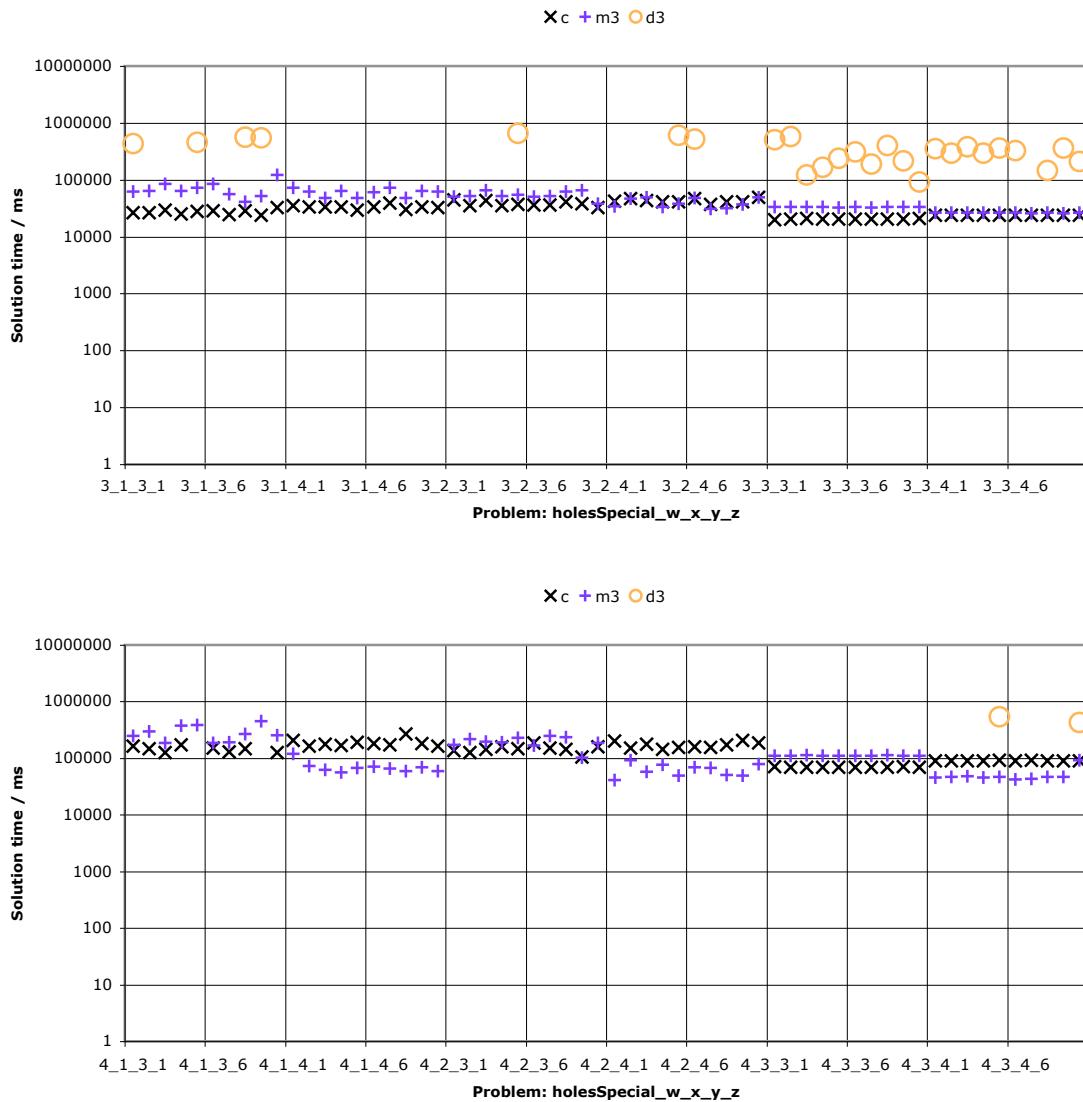


Figure 5.17: Average solution times for *holesSpecial* problems. Experiments are placed in groups of five for clarity. X axis labels refer to the problem name for the first experiment in each group: subsequent problems have successive *z* indices.

5.5 Summary of empirical analysis

This chapter presented experiments analysing the relative performance of the planning mechanisms from Chapter 3 and the planning algorithms from Chapter 4.

Comparison of planning mechanisms The experiments in Section 5.3 compared the pMPF and fMPF planning mechanisms from Chapter 3 on selected single agent first order recursive problems. As predicted in Section 3.5, the size of pMPF task trees was exponentially related to the number of world objects in the problem. The size of fMPF trees, by comparison, was linearly related to the number of world objects. Tree generation time was found to be primarily affected by the number of nodes in the tree: fMPF trees were found to be much smaller and faster to generate than pMPF trees despite the presence of planning variables.

The summary information in initial task trees was identical for the two formalisms. Planning with fMPF was found to be more memory efficient than planning with pMPF for the problems tested, partly because of the reduced maintenance time of the smaller trees, and partly because of the requirement for fewer iterations in problems of significant size. The lower number of iterations was attributed to the later binding of planning variables from methods in the problem description (Section 5.3).

Comparison of planning approaches As expected, the centralised planning and plan-then-merge approaches showed similar performance overall. Centralised planning faired better than plan-then-merge on problems that did not decompose naturally into smaller parts (such as many *bwRandom* problems), while plan-then-merge faired better on other problems that did (such as *bwSwap* problems). Unfortunately, “decomposability” could not be determined in advance from the structure of the task tree or summary information.

As expected, plan-then-merge was unable to solve many Navigation problems because of agents’ restricted control over resources (Section 5.4.2), but showed a surprising success at Holes problems because of individual agents’ ability to produce *abstract solutions* that could subsequently be merged into a valid joint plan (Section 5.4.3). The allocation of tasks to agents was also found to have a dramatic effect on plan-then-merge, partly because of the amount of

redundancy that needed to be removed during plan merging, and partly because of the effects of different task arrangements on planning (for example, in *bwReverse* problems). The restriction of knowledge of initial conditions was also useful, if used carefully in a problem specific way: inappropriate alteration of agents' knowledge has implications for the soundness of planning.

Distributed local planning was on the whole slower than the other approaches, and was subject to local search related problems such as *weight explosion* (Section 5.4.1) and *plateauing* (Section 5.4.2). Much of the trouble with the approach were attributed to the poor implementation of *nogood* constraints (Section 4.4.2), which did not effectively prevent agents from revisiting sets of plan features that had previously been found to be incompatible. Distributed local planning cannot be scaled to deal with larger problems in its current form: for this, significant reimplementations would be required.

Chapter 6 uses the the analysis above, along with insight gained through the rest of the thesis, to review the contributions from Section 1.3 and outline a course for future work.

Chapter 6

Conclusions and future work

6.1 Summary of contributions

The principal contributions of this thesis were outlined in Section 1.3. They are revisited here within the wider context of the whole thesis.

6.1.1 Taxonomy of multi agent planning problems

The space of multi agent planning problems is very large, far beyond the scope of this thesis. Section 1.1 provided a taxonomy of issues in multi agent planning, including a precise interpretation of multi agent planning, a set of standard features present in all multi agent planning problems according to this interpretation, and a discussion of features that may appear in some problems but not others.

The taxonomy is an important contribution for two reasons. Firstly, it proposes a set of standard definitions of terms that are used with a wide variety of meanings in the literature, removing ambiguity and providing a common language for talking about multi agent planning. Secondly, it creates a space of problems which researchers can use to categorise specific scenarios and identify the strengths and weaknesses of different planning approaches.

The most basic problem definition was chosen for study in the empirical section of this thesis, with just a few extensions selected from the taxonomy (Section 1.2). Essentially this reduces

to:

- producing fast, memory efficient multi agent planning that maximises the *independence* of the agents;
- implementing and evaluating multi agent planning with heterogeneous resource control;
- implementing and evaluating multi agent planning with heterogeneous knowledge about external state.

6.1.2 Approaches to multi agent planning

Rather than concentrating on a single approach to multi agent planning, the empirical work in this thesis attempted to compare and contrast different approaches on a variety of problems. Three approaches to multi agent planning were identified in Section 2.2: centralised planning, plan-then-merge and distributed local planning. Of these, centralised planning and plan-then-merge have been implemented extensively in the literature. Distributed local planning, however, has received relatively little attention, although extensive research has been done in the related fields of distributed constraint satisfaction (Yokoo and Hirayama, 2000) and single agent planning with local search (Kautz and Selman, 1992; Chien et al., 1999; Narayek, 2002).

The three approaches provide a representative selection of the variety of approaches that are possible. Most current multi agent planning systems follow one of these patterns. Notable simplifications in the approaches as presented included the following:

- Goal selection, assignment and reassignment were ignored. Agents were given their own set of goals that could not be changed or contracted out to others. This is in contrast with, for example, the work of Decker and Lesser (1992, 1995), which focuses on the distribution of problem solving processes between of agents, treating agents as resources for problem solving and data processing, across which processing may be load balanced.
- Consensus mechanisms such as negotiation were not used. Agents were considered to be self interested but trustworthy, goals were assumed to be non-conflicting, and partial solutions were not permitted, making distributed algorithms alone sufficient. It would be

possible to build negotiation in to any of the three approaches, expanding its applicability to a larger set of problems. This is an important future extension of this work.

- The planning system was not embedded in a more complete agent architecture. Other issues than planning, such as plan execution and learning from past mistakes, are important aspects of agency that need to be combined with planning to produce complete agents that can adapt to a wider range of problems from Section 1.1. Learning, in particular, is an important part of a more complete notion of agency.

6.1.3 Common planning mechanism for multi agent planning

The HTN planning mechanism MPF was developed in Chapter 3 to provide a common platform on which to implement the three multi agent planning approaches. MPF was based on the *CHiPs* planning mechanism (Clement and Durfee, 1999a,b,c; Clement, 2002). *CHiPs* and its task tree based plan representation and summary information based heuristics (Section 3.2) were a natural starting point for the development of MPF because of their previous use for the coordination of multiple plans and their foundation in standard HTN planning.

The empirical analysis in Section 5.3 indicates that MPF does not scale well when applied to standard planning domains such as Blocksworld and Navigation (Section 5.2). These are merely toy problems, intended as a means of investigating properties of the algorithms being developed: if an algorithm does not scale well on these it has little chance of faring well in the real world.

The inability to scale to larger problems is partly due to the time and memory taken maintaining multiple task trees, and partly due to the time taken calculating summary information for use in heuristics. The first of these problems was partly alleviated through the use of planning variables and value counting, which allowed larger problems to be tackled faster than propositional approaches. However, the maintenance of planning variables increased the amount of time spent recalculating summary information (Section 3.5) and doing tree simplification (Section 3.4.4). This made the evaluation of new partial plans much slower, which in turn slowed down the process of refinement and of planning overall. This meant that, while fMPF was still preferable to pMPF for most of the problems in this thesis, its use was not as beneficial as it

might otherwise have been.

While task trees and summary information have been shown to help the fast coordination of plans for multiple executives by a single *coordination agent* (Chapter 6 of Clement, 2002), and have been successfully implemented in the single agent local search planner ASPEN (Chapter 8 of Clement, 2002; Rabideau et al., 1999), they were not originally designed with decentralised planning in mind. MPF has several limitations as a result:

- The size of the initial task tree has to be calculated prior to planning. This is a simple task for non-recursive planning problems, but is more difficult when recursive tasks and methods are present. The value counting technique described in Section 3.6 is, in a sense, a naïve solution that is applicable in a limited set of cases. A more comprehensive solution involves allowing a small initial task tree to be expanded during planning (Gurnell, 2004)¹. This is the subject of ongoing research, and has not been discussed in the body of this thesis.
- Pure HTN planning proceeds by the decomposition of tasks. This is inappropriate for the implementation of plan merging and local search algorithms. As discussed in Section 4.3, plan merging ideally involves the addition of tasks to a plan as well as the detection and removal of conflicting or duplicate tasks. The implementation of this in MPF would involve adding new level 1 tasks to the task tree. However, such additions would require time consuming updates, would potentially cause a large increase in the size of search space, and would require lots of recalculation of summary based heuristics.
- Local search planning typically involves the use of special local search operators that repair flaws without having to use the normal refinement based search structure (Section 2.1.6). Again, this is difficult without dropping the decomposition-only planning paradigm and allowing the addition of tasks to the plan, either as forward- or backward-chaining state space search (Section 2.1.3) or in a least commitment style response to threats (Section 2.1.4). The distributed local search algorithm presented in Section 2.2.3 may benefit from the use of local search operators: currently it is “local” only because agents have local control over their own planning. Agents still perform a refinement based

¹Available from <http://www.cs.bham.ac.uk/~djg>.

search, interrupting it only when *reset* or *penalise* messages are received (Section 4.4.2). When search is interrupted it is reset to its initial state, which is potentially inefficient compared to more direct methods of threat resolution.

- The addition of *nogood* constraints is a fundamental requirement for completeness in distributed constraint satisfaction (Section 4.4.1). As discussed in Section 4.4.2, however, the implementation of nogood constraints in MPF is impractical as summary information changes whenever tasks are decomposed.

Many of the problems above are a result of the use of task trees in planning, as opposed to more the traditional plan representations described in Section 2.1. Plan merging and distributed local planning would be better suited to planning algorithms that do not rely on the storage of many copies of large, complicated plan structures and the continuous recalculation of expensive heuristics. While task trees support accurate heuristics that are useful for resolving inter-plan conflicts, such heuristics could also be generated from a number of other sources (Bonet and Geffner, 1998; Hawes, 2003; Hoffmann and Nebel, 2001; Nguyen et al., 2002).

6.1.4 Comparison of multi agent planning approaches

The multi agent planning approaches introduced in Section 2.2 and implemented in Chapter 4 were compared and contrasted in Section 5.4. The following observations were made:

Ability to solve problems Centralised planning was able to solve many of the problems attempted within the time and memory limits imposed. Plan-then-merge and distributed local planning could only solve subsets of these problems, the details of which are discussed in the relevant sections below.

Solution efficiency Centralised planning was faster than the other techniques on over half of all problems considered. Plan-then-merge was faster than centralised planning on a significant number of problems, and distributed local planning was generally not competitive with the others. This is perhaps to be expected, as the lower performance of distributed local planning must be offset against the increased independence it provides for the planning agents.

Quality of plans produced The heuristics in MPF favour plans with fewer threats between summary conditions: plan length and resource usage were not considered. Plans with fewer tasks tend to contain fewer conflicts, so near-optimal plans were common, but there was no guarantee of optimality.

It was hypothesised that plan-then-merge and distributed local planning may be at a disadvantage compared to centralised planning, because of the way in which they split search up into smaller parts. However, this was found not to be the case, as there were cases in which each algorithm performed best. Unfortunately, due to low solution rates in many problem types, trends and patterns in solution quality could not be analysed.

Plan-then-merge While the performance of plan-then-merge was good on problems it could solve, there were a large number of problems that it could not, three categories of which are identified below:

1. The individual planning stage often failed on problems in which agents had limited control of resources in the environment. For example, in *navLine* problems the movement of each robot apart from the leading robot required the movement of the robot in front: if an agent did not have control of the necessary robots it failed to produce an individual plan, and the whole team failed as a result.

This issue did not occur when agents had complete control over all resources in the problem. In *Blocksworld*, for example, every agent was able to move every block: agents simply moved blocks that prevented them achieving their goals in their individual plans, and redundant tasks were subsequently removed during plan merging.

The resource control issue can be overcome in certain cases by altering agents' knowledge of world state. For example, *navLine* problems are solvable if agents are unaware of the positions of robots that are not part of their individual plan (Section 5.4.2). In *navRing* problems, however, the restriction of knowledge did not help: non-mergeable plans were still produced. There was no general rule for when and how to restrict knowledge: restrictions were made in an ad hoc and problem specific way.

In *Holes* problems, planning agents were able to produce mergeable individual plans de-

spite having limited control of resources and knowledge of external state. This is because they were able to produce abstract solutions that represented a set of possible candidate plans: the structure of the Holes domain just so happened to encourage solutions that could be merged into a joint plan. While abstraction was helpful in this case, it is unlikely to occur in problems in general, especially if agents are only partially aware of the complete set of constraints on their tasks. Planning agents have to make some commitments to orderings, decompositions and variable bindings to produce valid individual plans: if they have limited control of resources or knowledge of the environment, it is likely that incompatible commitments will make plan merging impossible without subsequent re-planning.

2. The plan merging stage can fail if the merger does not have complete freedom to change agents' individual plans. As discussed previously, MPF does not permit the addition of tasks to the joint plan to remove conflicts. Every refinement planning algorithm has a restriction like this, because every refinement planning algorithm relies on backtracking to undo operations it has performed that have not led to a valid plan (Section 2.1.1). The trouble with this in plan merging is that some of the operations performed during planning may have to be undone to allow the plans to be merged. Local search techniques (Section 2.1.6) are not bound by this regimented refinement based approach, and may be better suited to the ad hoc nature of plan merging.
3. The plan merging stage can fail if agents create individual plans without complete knowledge of the environment. This was not an issue in Holes problems because of the ability of the agents to produce abstract solutions, but it did cause problems in *navRing* and some *navStar* problems.

Problems 1 and 3 above are a result of agents' isolation during the planning phase. These problems can be overcome in a number of ways:

1. Domain specific social rules can be imposed to prevent agents coming into conflict (Section 2.2.2; Shoham and Tennenholtz, 1995), although possibly at the expense of soundness.

2. Coordination can be performed prior to planning to ensure individual planning and merging stages are possible (Valk et al., 2005). If individual planning and/or merging are not possible, or if the coordination phase fails, centralised planning can be used as a fallback technique.
3. Agents can be allowed to communicate during planning, as in distributed local planning, the advantages and disadvantages of which are discussed below.

Distributed local planning Distributed local planning was outperformed by centralised planning and plan-then-merge on most problems. This is unsurprising due to the nature of the algorithm:

1. Because of the incompatibility of best first search (BFS) and the requirement for progressive commitment in distributed problem solving (Section 2.2.3), distributed local planning uses depth first search (DFS). Given the same heuristics, BFS will usually solve problems in at least as few refinements as DFS, so a single agent using distributed local planning will normally take longer than an equivalent agent using centralised planning, even without the presence of other agents to initiate *resets*. Distributed local planning only performs competitively with respect to plan-then-merge when the extra overhead of performing the distributed search with its depth first structure is less than the time taken to merge plans, and only performs competitively with respect to centralised planning if no *resets* and little backtracking is required.
2. Communication and updating of external summaries imposes a further overhead on agents using distributed local planning. Time taken evaluating summary based heuristics also contributes to this cost, as the “write” tokens used to avoid problems with concurrent communication (Section 4.4) have to be held while heuristics are calculated.
3. The algorithm used in distributed local planning is a cross between DFS and asynchronous backtracking (ABT, Yokoo and Hirayama, 2000, discussed in Section 4.4.1). In the case of unresolvable conflicts during planning, the ABT part of the algorithm interrupts the DFS part and *resets* the agent to its initial plan. Resetting is not an issue in distributed constraint satisfaction, as agents are only responsible for a single variable

each. However, a single reset in distributed local planning can force the agents involved to repeat many refinements that centralised planning and plan-then-merge agents would only perform once (Section 5.4.1).

4. Two aspects of distributed local planning meant that it fell foul of some of the classic problems of local search algorithms (Section 2.1.6). Firstly, the summary weights used to do the job of nogood constraints merely discouraged (rather than prevented) agents from revisiting combinations of plans that had previously caused unresolvable conflicts. It was possible for agents to get trapped in a local minimum from which they could not escape, creating a *weight explosion* (Section 5.4.2). Secondly, the policy of *waiting* for decompositions during third party threat resolution (Section 4.4.2) meant that agents could spend long periods of time on heuristic plateaux (Section 5.4.3). Although these plateaux did not cause search to fail, they imposed serious delays that caused time outs on many problems.

A number of techniques are suggested as workaround for these problems:

1. The completeness and speed of the ABT part of the distributed local planning algorithm are dependent on the use of concrete *nogood* constraints. The summary weighting technique described in Section 4.4.2 is a rudimentary approximation of the ABT technique, but suffers (end of Section 5.4.2) because it does not explicitly tell an agent which areas of search are not to be repeated after a reset. Real nogood constraints are required to avoid repeated resetting because of the same unresolvable conflicts.
2. Resets occur when unresolvable conflicts are found during planning. However, some unresolvable conflicts could be made resolvable by giving agents more flexible local search refinements and operators for plan repair.

Task trees and summary information are inappropriate for the implementation of nogood constraints and local search algorithms. Alternative planning mechanisms are required to fix these problems with distributed local planning.

6.1.5 Conclusions

Centralised planning is the fastest and most widely applicable approach above. This is unsurprising given that it has its roots in conventional single agent refinement planning, which has evolved over several decades. However, centralised planning does not provide agents in multi agent problems with any independence: it cannot be used when agents wish to maintain privacy, it permits misinterpretation of implicit or explicit goals (Section 1.1.3), it is inefficient when the overlap between agents' individual problems is small, and it will not scale up to large problems involving many agents and highly complex environments.

Plan-then-merge is representative of a variety of possible approaches based on plan merging. Plan merging approaches allow agents to retain a certain amount of independence while using the same single agent planning techniques. Consequently, if a problem lends itself well to decomposition, approaches like plan-then-merge can be faster than centralised planning in terms of both speed and memory usage. However, plan merging approaches can only be used when individual problems can be solved in isolation, and so their usefulness is restricted to a limited set of situations.

Distributed local planning is one of a variety of possible approaches in which agents can communicate, cooperate, exchange knowledge and exchange resources during planning. These algorithms are the most promising in terms of agent independence and their usefulness in real world multi agent environments, although they cannot be implemented with traditional refinement based planning techniques. Unfortunately, the implementation of distributed local planning algorithm in this thesis was quite inefficient, causing it to time out on many of the problems attempted. Further work is required on distributed local planning and other distributed planning techniques to enhance their performance and their applicability to larger, more complex planning problems. Lessons learned during the implementation and analysis of the algorithms in this thesis provide useful pointers for possible directions of future research.

6.2 Future work

As discussed above, distributed local planning is a promising approach despite a number of flaws in its implementation here. This section describes possibilities for the development of the approach into something more efficient and scalable:

Adoption of new planning mechanisms MPF is a complex and expensive planning mechanism: task trees are large structures that consume a lot of memory and take time to duplicate, and the calculation of summary based heuristics is expensive when first order trees are used. For distributed local planning to be effective, a simpler, more compact plan representation is needed.

One approach would be to encode planning as a constraint satisfaction problem (Chapter 8 of Ghallab et al., 2004; Frank et al., 2000). This would be convenient for the implementation of nogood constraints and other techniques inspired by distributed constraint satisfaction (DisCSP) problems (Section 4.4.1).

An alternative approach would be to use more traditional plan representations from plan space planning (Sections 2.1.4 and 2.1.5). If a pure HTN mechanism were used, agents would be restricted to planning by decomposition. This could be avoided by using a least commitment planning mechanism, or a hybrid HTN mechanism incorporating, for example, decomposition space and state space search (McCluskey et al., 2002). Nogood constraints could be represented as conjunctions of actions, binding constraints and ordering constraints, although care would need to be taken to preserve their meanings as actions are added to and deleted from the plan. Heuristics could be based on causal links (Section 2.1.4) and nogood constraints.

New distributed planning mechanisms and algorithms may benefit from ideas such as the exchange of external summary information, even if the word “*summary*” is not used with the same literal meaning it is here: the abstraction of relevant information for communication to other agents reduces communication costs and improves privacy and communications efficiency. Other key concepts include progressive commitment/decommitment from aspects of plans, continuous collection of nogood information, and other concepts from Section 4.4. Care must be taken when developing new planning mechanisms to satisfy the requirements discussed

in Section 3.1 without succumbing to the pitfalls observed in the empirical work in Chapter 5 (Section 6.1.4).

Adaptive approaches An important aspect of distributed local planning is that it is not *totally* reliant on communication to produce valid plans. Agents are not required to *wait* for new information to arrive: they continue planning regardless, and simply respond appropriately when external information does change. This should be an important consideration in any distributed planning algorithm: agents should be as independent as is sensible for any given problem and multi agent environment.

This thesis has demonstrated that different planning approaches are suited to different types of problem: sometimes a distributed approach is better than a centralised one and sometimes it is not. It was originally an intention to produce an adaptive planning system that automatically decided which of the three approaches to use and switch to it for the remainder of planning (Gurnell, 2003)². Unfortunately, the three approaches developed were different enough that the only sensible time to choose between them was before planning began. Summary information in the initial task tree proved to be uninformative enough to make such a decision impossible so early on in planning.

Within limits, distributed local planning can behave like centralised planning or distributed constraint satisfaction depending on the nature of the problem being solved. The important aspect of this is the number of conflicts that are discovered between agents' plans. By proactively changing the amount of communication between agents, it is hypothesised that a distributed planning algorithm can be produced that can move between two contrasting states:

Refinement driven search, in which communication is infrequent (and thus inexpensive), coordination is kept to a minimum, and refinement based planning dominates.

Coordination driven search, in which communication is frequent, coordination is maximised, and distributed constraint satisfaction style search dominates³.

By moving between these states appropriately, the three approaches of this thesis may be recre-

²Available from <http://www.cs.bham.ac.uk/~djg>.

³Local search operators (Section 2.1.6) may be used to minimise backtracking due to unresolvable inter-agent conflicts.

ated using a single algorithm. Appropriate points for switching could be identified by analysing the structure of the problem or the status of planning. If there is no reliable way to determine the best policy directly from the information available, agents could use learning techniques to identify situations that are similar to previously encountered problems and apply approaches that have been successful in the past. Research on phase transitions in constraint satisfaction and planning could also have relevance here (Section 5.5).

Application to wider classes of problem Real world multi agent planning problems may be large and complex, and may involve many difficult features not present in the toy problems in this thesis. If the efficiency of distributed planning techniques such as distributed local planning can be increased sufficiently, new classes of problem from Section 1.1 may be dealt with.

One class of problem is seen as particularly important: any generally applicable distributed planning algorithm must be able to cope with agents that have conflicting goals. As a last resort, agents should be able to resolve conflicts by dropping less important goals to produce a partial solution. Where agents are self interested, this will necessarily involve negotiation to ensure mutually satisfactory goal selection. Learning can also be important here: agents can use past experience to identify situations where compromises might be necessary and come up with sensible strategies with little or no search.

Distributed local planning, while currently unperfected and significantly slower than centralised planning and plan merging approaches, holds the key to the application of planning techniques to a wider class of social situations. If agents can be developed that are able to plan satisfactorily when other agents are present, with problems of varying complexity and decomposability and potentially conflicting goals, then a significant step will have been taken towards generalising planning into an activity where agents can take a wide set of multi agent problems in their stride.

Appendix A

Guide to notation

This appendix briefly describes some of the non-standard text formatting and notation used in the descriptions of algorithms and data structures in Chapters 2 to 4.

A.1 Data structures

Names of data structures The names of data structures, actions, state literals and world objects are written using *mathematical italics*, as are the names of their types.

Names of planning variables The names of planning variables (variables created by a planner to refer to a disjunction of possible world objects) are prefixed with a question mark. For example: $?src$, $?x$, $?r_1$ and so on.

Names of types of world object The names of types of world objects in *pMPF* are prefixed with an exclamation mark. For example: $!location$, $!robot$, $!block$ and so on.

Tuples Most data structures are defined as tuples, the elements of which are written between angular brackets (“ \langle ” and “ \rangle ”). For example:

$$\langle t_1, t_2, t_3 \dots t_n \rangle$$

Accessor functions Wherever a tuple is defined, accessor functions are implicitly defined using

the names of its elements written in *mathematical italics*. For example, the definition above implies the following accessors for a tuple t :

$$\begin{aligned} &t_1(t) \\ &t_2(t) \\ &t_3(t) \\ &\vdots \\ &t_n(t) \end{aligned}$$

Names of functions, refinements and operators The names of all other subroutines, functions, refinements and planning operators are written as textual mathematical operators.

A.2 Pseudocode

Block structure Block structure is indicated by indentation, in a similar fashion to the Python programming language.

Keywords Keywords for programming constructs are written in **bold text**.

Assignments Assignments are written as left hand arrows (“←”). For example:

$$x \leftarrow x + 1$$

Assignments with forward propagation The double left arrow symbol (“⇐”) is used to denote the addition of a constraint to a (temporal or binding) constraint network, followed by forward propagation to update the state of the network accordingly. For example:

$$network \Leftarrow constraint$$

Function headers Each function takes zero or more arguments and returns zero or more return values. The return values of a function, if any, are written after a right hand arrow (“→”) in the function header. For example:

```
1 function my_function(a, b) → (c, d) :  
2     line 1  
3     line 2  
4     line 3  
5     etc...
```

Returning from functions Functions *always* return the specified number of values. This either occurs when function execution ends naturally, or when a **return** statement is encountered. The values returned are always the values with the names specified in the function header.

Appendix B

Domain descriptions and sample problems

This appendix presents the domain description and sample problem description files for each of the three test domains in Chapter 5. While the XML syntax used has not been formally introduced in the body of this thesis, readers of Chapter 3 should find it simple and intuitive.

The `planner=""` attributes of `task` nodes in the sample problems are used to identify groups of tasks, state constraints and methods belonging to the same agent. These attributes are mapped to actual agents during task tree generation. Top level state constraints that do not have `planner=""` attributes are global constraints that are part of all agents' problems. Variable identifiers are denoted by initial question marks. Type identifiers are denoted by initial exclamation marks.

B.1 Blocksworld

B.1.1 Domain description

```
<?xml version="1.0" encoding="utf-8"?>
<domain name="blocksworld">
  <type name="!surface">
  </type>
  <type name="!block">
    <extends>!surface</extends>
  </type>
  <type name="!table">
    <extends>!surface</extends>
  </type>
</domain>
```



```

    <values>table</values>
</type>

<predicate>on(!block !surface)</predicate>
<predicate>clear(!surface)</predicate>

<action type="primitive">move(!block !surface !surface)</action>
<action type="abstract">achieve_clear(!block)</action>
<action type="abstract">achieve_on(!block !surface)</action>

<method name="move_block_block" action="move(?block ?src ?des)">
  <var name="?block" type="!block"/>
  <var name="?src" type="!block"/>
  <var name="?des" type="!block"/>

  <notequal a="?block" b="?src"/>
  <notequal a="?block" b="?des"/>
  <notequal a="?src" b="?des"/>

  <pre name="pre1">clear(?block)</pre>
  <pre name="pre2">clear(?des)</pre>
  <pre name="pre3">on(?block ?src)</pre>

  <post name="post1">not (on(?block ?src))</post>
  <post name="post2">clear(?src)</post>
  <post name="post3">on(?block ?des)</post>
  <post name="post4">not (clear(?des))</post>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="pre3">e</order>
  <order a="pre1" b="post1">m</order>
  <order a="post1" b="post2">m</order>
  <order a="post1" b="post3">m</order>
  <order a="post1" b="post4">m</order>
</method>

<method name="move_block_table" action="move(?block ?src ?des)">
  <var name="?block" type="!block"/>
  <var name="?src" type="!block"/>
  <var name="?des" type="!table"/>

  <notequal a="?block" b="?src"/>
  <notequal a="?block" b="?des"/>
  <notequal a="?src" b="?des"/>

  <pre name="pre1">clear(?block)</pre>
  <pre name="pre2">on(?block ?src)</pre>

  <post name="post1">not (on(?block ?src))</post>
  <post name="post2">clear(?src)</post>
  <post name="post3">on(?block ?des)</post>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="post1">m</order>
  <order a="post1" b="post2">m</order>
  <order a="post1" b="post3">m</order>
</method>

<method name="move_table_block" action="move(?block ?src ?des)">
  <var name="?block" type="!block"/>
  <var name="?src" type="!table"/>
  <var name="?des" type="!block"/>

  <notequal a="?block" b="?src"/>
  <notequal a="?block" b="?des"/>
  <notequal a="?src" b="?des"/>

  <pre name="pre1">clear(?block)</pre>
  <pre name="pre2">clear(?des)</pre>
  <pre name="pre3">on(?block ?src)</pre>

  <post name="post1">not (on(?block ?src))</post>
  <post name="post2">on(?block ?des)</post>
  <post name="post3">not (clear(?des))</post>

```

```

    <order a="pre1" b="pre2">e</order>
    <order a="pre2" b="pre3">e</order>
    <order a="pre1" b="post1">m</order>
    <order a="post1" b="post2">m</order>
    <order a="post1" b="post3">m</order>
</method>

<method name="achieve_clear" action="achieve_clear(?block)">
  <var name="?block" type="!block"/>
  <var name="?on" type="!block"/>
  <var name="?table" type="!table"/>

  <notequal a="?block" b="?on"/>

  <task name="task1">achieve_clear(?on)</task>
  <task name="task2">move(?on ?block ?table)</task>

  <order a="task1" b="task2">b m</order>
</method>

<method name="achieve_clear_noop" action="achieve_clear(?block)">
  <var name="?block" type="!block"/>

  <pre name="pre1">clear(?block)</pre>
</method>

<method name="achieve_on" action="achieve_on(?block ?des)">
  <var name="?block" type="!block"/>
  <var name="?src" type="!surface"/>
  <var name="?des" type="!surface"/>

  <notequal a="?block" b="?src"/>
  <notequal a="?block" b="?des"/>
  <notequal a="?src" b="?des"/>

  <task name="task1">achieve_clear(?block)</task>
  <task name="task2">achieve_clear(?des)</task>

  <task name="task3">move(?block ?src ?des)</task>

  <order a="task1" b="task3">b</order>
  <order a="task2" b="task3">b</order>
</method>

<method name="achieve_on_noop" action="achieve_on(?block ?des)">
  <var name="?block" type="!block"/>
  <var name="?des" type="!surface"/>

  <notequal a="?block" b="?des"/>

  <pre name="pre1">on(?block ?des)</pre>
</method>
</domain>

```

B.1.2 Sample problem: bwReverse_3

```

<?xml version="1.0" encoding="utf-8"?>
<problem name="bwReverse_3" domain="blocksworld">
  <type name="!block">
    <values>a b c</values>
  </type>
  <global>
    <post name="post1">clear(c)</post>
    <post name="post2">on(c b)</post>
    <post name="post3">on(b a)</post>
  </global>
</problem>

```

```

<post name="post4">on(a table)</post>

<task name="task1" planner="p11">achieve_on(a b)</task>
<task name="task2" planner="p12">achieve_on(b c)</task>

<pre name="pre1" planner="p11">on(a b)</pre>
<pre name="pre2" planner="p12">on(b c)</pre>

<order a="post1" b="post2">e</order>
<order a="post1" b="post3">e</order>
<order a="post1" b="post4">e</order>

<order a="post1" b="task1">b m</order>
<order a="post1" b="task2">b m</order>

<order a="pre1" b="pre2">e</order>

<order a="task1" b="pre1">b m</order>
<order a="task2" b="pre1">b m</order>
</global>

<planner name="p11">
  <methods>
    achieve_on
    achieve_on_noop
    achieve_clear
    achieve_clear_noop
    move_block_block
    move_block_table
    move_table_block
  </methods>
</planner>

<planner name="p12">
  <methods>
    achieve_on
    achieve_on_noop
    achieve_clear
    achieve_clear_noop
    move_block_block
    move_block_table
    move_table_block
  </methods>
</planner>
</problem>

```

B.2 Navigation

B.2.1 Domain description

```

<?xml version="1.0" encoding="utf-8"?>
<domain name="navigation">
  <type name="!loc">
</type>

  <type name="!robot">
</type>

  <predicate>edge(!loc !loc)</predicate>
  <predicate>at(!robot !loc)</predicate>
  <predicate>clear(!loc)</predicate>

  <action type="primitive">move(!robot !loc !loc)</action>

```

```

<action type="abstract">travel(!robot !loc !loc)</action>

<method name="travel_indirect" action="travel(?r ?src ?des)">
  <var name="?r" type="!robot"/>
  <var name="?src" type="!loc"/>
  <var name="?aux" type="!loc"/>
  <var name="?des" type="!loc"/>

  <notequal a="?src" b="?aux"/>

  <task name="task1">move(?r ?src ?aux)</task>
  <task name="task2">travel(?r ?aux ?des)</task>

  <order a="task1" b="task2">b m</order>
</method>

<method name="travel_direct" action="travel(?r ?src ?des)">
  <var name="?r" type="!robot"/>
  <var name="?src" type="!loc"/>
  <var name="?des" type="!loc"/>

  <notequal a="?src" b="?des"/>

  <task name="task1">move(?r ?src ?des)</task>
</method>

<method name="move" action="move(?r ?a ?b)">
  <var name="?r" type="!robot"/>
  <var name="?a" type="!loc"/>
  <var name="?b" type="!loc"/>

  <notequal a="?a" b="?b"/>

  <pre name="pre1">edge(?a ?b)</pre>
  <pre name="pre2">at(?r ?a)</pre>
  <pre name="pre3">clear(?b)</pre>

  <post name="post1">at(?r ?b)</post>
  <post name="post2">not clear(?b)</post>
  <post name="post3">not at(?r ?a)</post>
  <post name="post4">clear(?a)</post>

  <order a="pre1" b="pre2">si</order>
  <order a="pre2" b="pre3">si</order>

  <order a="pre1" b="post1">fi</order>

  <order a="pre2" b="post1">o</order>
  <order a="pre3" b="post1">m</order>

  <order a="post1" b="post2">e</order>
  <order a="post1" b="post3">e</order>
  <order a="post1" b="post4">e</order>
</method>

</domain>

```

B.2.2 Sample problem: navRing_3_2

```

<?xml version="1.0" encoding="utf-8"?>
<problem name="navRing_3_2" domain="navigation">
  <type name="!robot">
    <values>r1 r2</values>
  </type>

  <type name="!loc">
    <values>p1 p2 p3</values>
  </type>

```

```

<global>
  <post name="post1">at (r1 p2)</post>
  <post name="post2">at (r2 p1)</post>

  <post name="post3">clear (p3)</post>

  <post name="post4">edge (p1 p2)</post>
  <post name="post5">edge (p2 p1)</post>
  <post name="post6">edge (p2 p3)</post>
  <post name="post7">edge (p3 p2)</post>
  <post name="post8">edge (p3 p1)</post>
  <post name="post9">edge (p1 p3)</post>

  <task name="task1" planner="p11">travel (r1 p2 p1)</task>
  <task name="task2" planner="p12">travel (r2 p1 p2)</task>

  <pre name="pre1" planner="p11">at (r1 p1)</pre>
  <pre name="pre2" planner="p12">at (r2 p2)</pre>

  <order a="post1" b="post2">e</order>
  <order a="post1" b="post3">e</order>
  <order a="post1" b="post4">e</order>
  <order a="post1" b="post5">e</order>
  <order a="post1" b="post6">e</order>
  <order a="post1" b="post7">e</order>
  <order a="post1" b="post8">e</order>
  <order a="post1" b="post9">e</order>

  <order a="post1" b="task1">b</order>
  <order a="post1" b="task2">b</order>

  <order a="task1" b="pre1">b</order>
  <order a="task2" b="pre1">b</order>

  <order a="pre1" b="pre2">e</order>
</global>

<planner name="p11">
  <methods>travel_indirect travel_direct move</methods>
</planner>

<planner name="p12">
  <methods>travel_indirect travel_direct move</methods>
</planner>
</problem>

```

B.3 Holes

B.3.1 Domain description

```

<?xml version="1.0" encoding="utf-8"?>
<domain name="holes">
  <type name="!object">
  </type>

  <type name="!peg">
    <extends>!object</extends>
  </type>

  <type name="!hole">
    <extends>!object</extends>
  </type>

```

```

<type name="!feature">
  <values>any</values>
</type>

<type name="!shape">
  <extends>!feature</extends>
</type>

<type name="!color">
  <extends>!feature</extends>
</type>

<type name="!size">
  <extends>!feature</extends>
</type>

<predicate type="dynamic">empty(!hole)</predicate>
<predicate type="dynamic">placed(!peg)</predicate>
<predicate type="dynamic">in(!peg !hole)</predicate>
<predicate type="static">shape(!object !shape)</predicate>
<predicate type="static">size(!object !size)</predicate>
<predicate type="static">color(!object !color)</predicate>

<action type="abstract">locate(!peg)</action>
<action type="primitive">place(!peg !hole)</action>

<method name="match_all" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?shape" type="!shape"/>
  <var name="?size" type="!size"/>
  <var name="?color" type="!color"/>

  <pre name="pre1">shape(?peg ?shape)</pre>
  <pre name="pre2">shape(?hole ?shape)</pre>
  <pre name="pre3">size(?peg ?size)</pre>
  <pre name="pre4">size(?hole ?size)</pre>
  <pre name="pre5">color(?peg ?color)</pre>
  <pre name="pre6">color(?hole ?color)</pre>

  <task name="task1">place(?peg ?hole)</task>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="pre3">e</order>
  <order a="pre1" b="pre4">e</order>
  <order a="pre1" b="pre5">e</order>
  <order a="pre1" b="pre6">e</order>
  <order a="pre1" b="task1">m</order>
</method>

<method name="match_shape" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?shape" type="!shape"/>

  <pre name="pre1">not placed(?peg)</pre>
  <pre name="pre1">shape(?peg ?shape)</pre>
  <pre name="pre2">shape(?hole ?shape)</pre>

  <task name="task1">place(?peg ?hole)</task>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="task1">m</order>
</method>

<method name="verify_shape" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?shape" type="!shape"/>

```

```

    <pre name="pre1">in(?peg, ?hole)</pre>
    <pre name="pre2">shape(?peg ?shape)</pre>
    <pre name="pre3">shape(?hole ?shape)</pre>

    <order a="pre1" b="pre2">e</order>
    <order a="pre1" b="pre3">e</order>
</method>

<method name="match_size" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?size" type="!size"/>

  <pre name="pre1">size(?peg ?size)</pre>
  <pre name="pre2">size(?hole ?size)</pre>

  <task name="task1">place(?peg ?hole)</task>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="task1">m</order>
</method>

<method name="verify_size" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?size" type="!size"/>

  <pre name="pre1">in(?peg, ?hole)</pre>
  <pre name="pre2">size(?peg ?size)</pre>
  <pre name="pre3">size(?hole ?size)</pre>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="pre3">e</order>
</method>

<method name="match_color" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?color" type="!color"/>

  <pre name="pre1">color(?peg ?color)</pre>
  <pre name="pre2">color(?hole ?color)</pre>

  <task name="task1">place(?peg ?hole)</task>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="task1">m</order>
</method>

<method name="verify_color" action="locate(?peg)">
  <var name="?peg" type="!peg"/>

  <var name="?hole" type="!hole"/>
  <var name="?color" type="!color"/>

  <pre name="pre1">in(?peg, ?hole)</pre>
  <pre name="pre2">color(?peg ?color)</pre>
  <pre name="pre3">color(?hole ?color)</pre>

  <order a="pre1" b="pre2">e</order>
  <order a="pre1" b="pre3">e</order>
</method>

<method name="place" action="place(?peg ?hole)">
  <var name="?peg" type="!peg"/>
  <var name="?hole" type="!hole"/>

  <pre name="pre1">empty(?hole)</pre>
  <pre name="pre1">not_placed(?peg)</pre>

  <post name="post1">not_empty(?hole)</post>

```

```

    <post name="post2">placed(?peg) </post>
    <post name="post3">in(?peg ?hole) </post>

    <order a="pre1" b="post1">m</order>
    <order a="post1" b="post2">e</order>
    <order a="post1" b="post3">e</order>
  </method>

</domain>

```

B.3.2 Sample problem: holesSpecial_3_1_3_1

```

<?xml version="1.0" encoding="utf-8"?>
<problem name="holesSpecial_3_1_3_1" domain="holes">

  <type name="!peg">
    <values>b1 b2 b3</values>
  </type>

  <type name="!hole">
    <values>h1 h2 h3</values>
  </type>

  <type name="!shape">
    <values>circle triangle square</values>
  </type>

  <type name="!color">
    <values>red yellow pink</values>
  </type>

  <type name="!size">
    <values>medium large small</values>
  </type>

  <global>
    <post name="post1">shape(b1 triangle) </post>
    <post name="post2">color(b1 red) </post>
    <post name="post3">size(b1 large) </post>

    <post name="post4">shape(b2 circle) </post>
    <post name="post5">color(b2 pink) </post>
    <post name="post6">size(b2 large) </post>

    <post name="post7">shape(b3 square) </post>
    <post name="post8">color(b3 red) </post>
    <post name="post9">size(b3 medium) </post>

    <post name="post10">shape(h1 triangle) </post>
    <post name="post11">color(h1 red) </post>
    <post name="post12">color(h1 pink) </post>
    <post name="post13">size(h1 large) </post>
    <post name="post14">size(h1 medium) </post>
    <post name="post15">empty(h1) </post>

    <post name="post16">shape(h2 triangle) </post>
    <post name="post17">shape(h2 circle) </post>
    <post name="post18">shape(h2 square) </post>
    <post name="post19">color(h2 pink) </post>
    <post name="post20">size(h2 large) </post>
    <post name="post21">size(h2 medium) </post>
    <post name="post22">empty(h2) </post>

    <post name="post23">shape(h3 square) </post>
    <post name="post24">color(h3 red) </post>
    <post name="post25">size(h3 large) </post>
    <post name="post26">size(h3 medium) </post>
    <post name="post27">empty(h3) </post>
  </global>

```



```

<task name="task1" planner="p11">locate (b1) </task>
<task name="task2" planner="p12">locate (b1) </task>
<task name="task3" planner="p13">locate (b1) </task>
<task name="task4" planner="p11">locate (b2) </task>
<task name="task5" planner="p12">locate (b2) </task>
<task name="task6" planner="p13">locate (b2) </task>
<task name="task7" planner="p11">locate (b3) </task>
<task name="task8" planner="p12">locate (b3) </task>
<task name="task9" planner="p13">locate (b3) </task>

<order a="post1" b="post2">e</order>
<order a="post1" b="post3">e</order>
<order a="post1" b="post4">e</order>
<order a="post1" b="post5">e</order>
<order a="post1" b="post6">e</order>
<order a="post1" b="post7">e</order>
<order a="post1" b="post8">e</order>
<order a="post1" b="post9">e</order>
<order a="post1" b="post10">e</order>
<order a="post1" b="post11">e</order>
<order a="post1" b="post12">e</order>
<order a="post1" b="post13">e</order>
<order a="post1" b="post14">e</order>
<order a="post1" b="post15">e</order>
<order a="post1" b="post16">e</order>
<order a="post1" b="post17">e</order>
<order a="post1" b="post18">e</order>
<order a="post1" b="post19">e</order>
<order a="post1" b="post20">e</order>
<order a="post1" b="post21">e</order>
<order a="post1" b="post22">e</order>
<order a="post1" b="post23">e</order>
<order a="post1" b="post24">e</order>
<order a="post1" b="post25">e</order>
<order a="post1" b="post26">e</order>
<order a="post1" b="post27">e</order>

<order a="post1" b="task1">b m</order>
<order a="post1" b="task2">b m</order>
<order a="post1" b="task3">b m</order>
<order a="post1" b="task4">b m</order>
<order a="post1" b="task5">b m</order>
<order a="post1" b="task6">b m</order>
<order a="post1" b="task7">b m</order>
<order a="post1" b="task8">b m</order>
<order a="post1" b="task9">b m</order>
</global>

<planner name="p11">
  <methods>match_shape verify_shape place</methods>
</planner>

<planner name="p12">
  <methods>match_color verify_color place</methods>
</planner>

<planner name="p13">
  <methods>match_size verify_size place</methods>
</planner>
</problem>

```

Appendix C

Experimental data

The tables of timing data in Section 5.4 only include information on the number of successful runs and the average time taken to solve the problem with each algorithm. This appendix contains more detailed information on:

- The percentage success, average solution time and average number of refinements for runs of the centralised planning algorithm.
- The percentage success, average solution time, average merging time and average number of refinements for runs of the plan-then-merge algorithm.
- The percentage success, average solution time, average number of refinements and average number of resets for runs of the distributed local planning algorithm.

Columns are labelled according to the planning algorithm and number of agents involved:

- c indicates centralised planning;
- mn indicates plan-then-merge with n agents;
- dn indicates distributed local planning with n agents.

Time to generate and download initial task trees and upload final plans, including individual plans during plan-then-merge, is ignored. Time spent communicating external summary information in distributed local planning, however, is included. Numbers of refinements in distributed local planning include “wait” refinements where one agent is waiting for another to decompose an appropriate effector (Section 4.4.2).

The following special codes are used where the above fields cannot be displayed:

- a blank entry indicates that an algorithm was not run on a problem;
- *t/out (p)* indicates that an algorithm timed out during planning in every run;
- *t/out (m)* indicates that an algorithm timed out during plan merging in every run;
- *fail (p)* indicates that an algorithm exhausted its search during planning in every run;
- *fail (m)* indicates that an algorithm exhausted its search during plan merging in every run.

Table C.1: Complete experimental data for *bwSwap* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwSwap_4</i>	100% 13574 38	100% 3773 1323 24			100% 8874 41 0		
<i>bwSwap_6</i>	100% 391600 172	80% 39198 2700 48	100% 6496 2716 33		100% 194844 604 0	100% 64870 29 0	t/out (p)
<i>bwSwap_8</i>	t/out (p)	80% 94671 4666 72	100% 94022 4692 51	100% 11388 4310 39	t/out (p)	t/out (p)	t/out (p)
<i>bwSwap_10</i>	t/out (p)	t/out (p)	60% 210204 7367 69	40% 215549 7454 57	t/out (p)	t/out (p)	t/out (p)

Table C.2: Complete experimental data for *bwReverseRobin* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwReverseRobin_2</i>	100% 441 3						
<i>bwReverseRobin_3</i>	100% 2295 16	100% 3207 1907 26			100% 4054 15 0		
<i>bwReverseRobin_4</i>	100% 6360 24	100% 8953 6146 72	t/out (m)	t/out (m)	100% 13244 35 0	100% 64630 43 5	
<i>bwReverseRobin_5</i>	100% 16812 31	40% 86622 44619 244	80% 32430 28498 105		100% 33419 22 0	100% 49943 25 0	t/out (p)
<i>bwReverseRobin_6</i>	100% 45553 39	t/out (p)	t/out (p)	40% 240317 231156 296	100% 84071 69 0	100% 116066 21 0	20% 258353 26 0
<i>bwReverseRobin_7</i>	100% 129446 48	t/out (p)	t/out (p)	t/out (m)	100% 228933 63 0	t/out (p)	t/out (p)
<i>bwReverseRobin_8</i>	100% 339387 56	t/out (p)	t/out (p)	t/out (p)	100% 495945 119 0	t/out (p)	t/out (p)
<i>bwReverseRobin_9</i>	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseRobin_10</i>	100% 441 3						
<i>bwReverseSeq_2</i>	100% 2295 16	40% 2465 1060 16			100% 7453 17 0		
<i>bwReverseSeq_3</i>	100% 6360 24	40% 5766 3875 36	40% 5496 3512 27		100% 40472 39 0	100% 60339 23 1	
<i>bwReverseSeq_4</i>	100% 16812 31	40% 13614 8539 45	40% 12521 9062 47	t/out (m)	100% 102875 23 0	100% 475733 67 7	40% 387466 39 0
<i>bwReverseSeq_5</i>	100% 45553 39	40% 25341 17340 62	40% 49694 45765 98	t/out (m)	100% 452123 106 20	40% 624514 48 10	t/out (p)
<i>bwReverseSeq_6</i>	100% 129446 48	40% 41671 21210 64	40% 125744 113879 131	t/out (m)	80% 590276 32 0	t/out (p)	t/out (p)
<i>bwReverseSeq_7</i>	100% 339387 56	40% 72795 43572 78	40% 241303 224289 156	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_8</i>	t/out (p)	t/out (m)	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_9</i>	t/out (p)	t/out (m)	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwReverseSeq_10</i>	t/out (p)	t/out (m)	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)

Table C.3: Complete experimental data for *bwRandom* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwRandom_4_2</i>	100% 2558 8	100% 2528 1145 17	100% 2242 784 13	100% 2453 1273 16	100% 4125 7 0	100% 5618 5 0	100% 11889 4 0
<i>bwRandom_4_3</i>	100% 118833 186	t/out (p)	100% 6648 3116 25	t/out (m)	t/out (p)	t/out (p)	t/out (p)
<i>bwRandom_4_4</i>	100% 4480 12	100% 4190 2156 24	100% 4018 2356 24	100% 4971 2151 22	100% 11430 6 0	100% 11358 5 0	100% 19632 3 0
<i>bwRandom_4_5</i>	100% 4906 15	100% 7958 4464 29	100% 24728 17907 109	t/out (m)	100% 18766 38 0	100% 31459 21 0	100% 43032 21 0
<i>bwRandom_4_6</i>	100% 297896 776	100% 48138 9094 88	100% 32760 25925 74	100% 33376 26304 75	80% 391353 221 46	100% 160838 31 3	100% 234950 27 2
<i>bwRandom_4_7</i>	100% 6512 18	100% 8032 4261 42	100% 8698 6321 44	100% 8534 5998 42	100% 40222 44 2	100% 38672 18 1	100% 86273 18 2
<i>bwRandom_4_8</i>	100% 7286 17	100% 13008 1357 24	100% 2510 1253 18	100% 2456 1315 18	100% 19651 8 0	100% 9010 5 0	100% 11651 4 0
<i>bwRandom_4_9</i>	100% 6963 21	100% 13898 7082 36	100% 19088 15606 63	100% 19349 15677 61	100% 69792 58 7	100% 44140 16 0	100% 87729 16 2
<i>bwRandom_4_10</i>	100% 123348 356	100% 188251 11193 260	t/out (m)	t/out (m)	t/out (p)	100% 76188 16 0	100% 99904 17 0
<i>bwRandom_4_11</i>	100% 7659 22	100% 10422 5100 32	100% 14506 10532 50	t/out (m)	100% 34721 47 0	100% 49821 18 0	100% 124590 32 2
<i>bwRandom_4_12</i>	100% 2914 9	100% 3191 1373 18	100% 6416 1774 17	100% 3305 1430 16	100% 6909 32 0	100% 11372 6 0	100% 17636 4 0
<i>bwRandom_4_13</i>	100% 10407 27	100% 9455 5057 36	100% 9477 5164 30	100% 12556 8793 36	100% 39633 35 7	100% 50072 12 0	80% 228519 28 5
<i>bwRandom_4_14</i>	t/out (p)	t/out (m)	100% 21300 14443 43	t/out (m)	100% 160094 90 13	100% 126994 46 5	100% 174400 33 1
<i>bwRandom_4_15</i>	100% 7527 19	100% 22916 4264 42	100% 10640 6428 34	t/out (m)	100% 18043 12 0	100% 45318 11 0	100% 121318 21 4

Problem	c	m2	m3	m4	d2	d3	d4
<i>bwRandom-4-16</i>	3729 11 100%	7308 1697 22 100%	4422 1730 19 100%	4361 1646 19 100%	9212 21 0	15796 10 0	23795 11 0
<i>bwRandom-4-17</i>	33131 109 100%	t/out (m)	t/out (m)	t/out (m)	t/out (p)	100% 314004 57 6	100% 417068 50 7
<i>bwRandom-4-18</i>	52626 92 100%	5659 2270 22 100%	t/out (m)	t/out (m)	t/out (p)	100% 69144 41 0	t/out (p)
<i>bwRandom-4-19</i>	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)
<i>bwRandom-4-20</i>	154815 273 100%	7038 2104 24 100%	t/out (m)	t/out (m)	t/out (p)	t/out (p)	t/out (p)

Table C.4: Complete experimental data for *navLine* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navLine-3-1</i>	579 2 100%	838 382 6 100%			1114 2 0 100%		
<i>navLine-3-2</i>	579 3 100%	2215 1055 12 100%			2717 5 0 100%		
<i>navLine-4-1</i>	1402 3 100%	1257 642 8 100%	1366 722 8		1983 3 0 100%	2242 2 0	
<i>navLine-4-2</i>	2067 6 100%	6332 3671 29 100%	4989 2976 17 100%		98392 107 61 100%		
<i>navLine-4-3</i>	1068 5 100%	3704 1932 18 100%	1820 1085 11 100%	2147 1289 10 100%	8197 15 0 100%	12333 7 0	
<i>navLine-5-1</i>	2559 4 100%	2136 1129 13 100%			2939 5 0 100%	5837 3 0 40%	8999 2 0
<i>navLine-5-2</i>	5528 9 100%	16610 9240 42 100%			283050 121 99 100%		
<i>navLine-5-3</i>	5717 10 100%	14035 9259 50 100%	15177 11240 33 100%		30121 33 0 100%	t/out (p)	
<i>navLine-5-4</i>	1728 7 100%	9925 4391 24 100%	7459 5174 21 100%	9528 6906 20 20%	28385 10 0 100%	68449 9 0 100%	87357 6 0
<i>navLine-6-1</i>	5235 5 100%	3252 1893 16 100%	2891 2171 14 100%	15004 10604 64 100%	7409 5 0 100%	14262 5 0 100%	15886 3 0

Table C.5: Complete experimental data for *navRing* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navRing-3-2</i>	966 6 100%	fail (m)			2105 4 0 100%		
<i>navRing-4-1</i>	987 2 100%	fail (m)			28274 32 6 100%		
<i>navRing-4-2</i>	2103 6 100%	fail (m)	fail (m)		t/out (p)	133506 36 4	
<i>navRing-4-3</i>	9640 41 100%	100% 4879 2132 18 100%	fail (m)		7605 10 0 100%		
<i>navRing-5-1</i>	1332 2 100%	fail (m)	fail (m)		t/out (p)	39137 18 7	
<i>navRing-5-2</i>	3855 9 100%	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navRing-5-3</i>	197350 373 100%	fail (m)	fail (m)		9884 10 0 100%		
<i>navRing-5-4</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	433342 39 18	
<i>navRing-6-1</i>	3009 3 100%	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navRing-6-2</i>	9950 9 100%	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navRing-6-3</i>	453511 337 100%	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navRing-6-4</i>	t/out (p)	t/out (m)	0 0 0	t/out (m)	t/out (p)	t/out (p)	
<i>navRing-6-5</i>	t/out (p)	t/out (p)	t/out (p)	t/out (m)	t/out (p)	t/out (p)	

Table C.6: Complete experimental data for *navStar* problems.

Problem	c	m2	m3	m4	d2	d3	d4
<i>navStar-3-2-1</i>	100% 28730 18	fail (m)			100% 104595 55 0		
<i>navStar-3-2-2</i>	100% 367634 290	fail (m)			80% 182565 156 20		
<i>navStar-3-2-3</i>	100% 81359 61	fail (m)			t/out (p)		
<i>navStar-3-2-4</i>	100% 160218 165	fail (m)			20% 53984 32 14		
<i>navStar-3-2-5</i>	100% 15090 9	9728 885 12			100% 66902 99 0		
<i>navStar-3-2-6</i>	100% 7072 5	8956 885 12			100% 66697 100 0		
<i>navStar-3-2-7</i>	100% 8846 6	7034 1188 12			100% 13076 14 0		
<i>navStar-3-2-8</i>	100% 233345 194	fail (m)			t/out (p)		
<i>navStar-3-2-9</i>	100% 22882 18	fail (m)			100% 17354 16 0		
<i>navStar-3-2-10</i>	100% 177119 103	fail (m)			100% 82791 75 0		
<i>navStar-4-2-1</i>	100% 25190 7	16001 2397 18			100% 261841 92 0		
<i>navStar-4-2-2</i>	100% 9295 5	9249 1059 12			100% 31836 47 0		
<i>navStar-4-2-3</i>	80% 487028 133	fail (m)			80% 537830 238 58		
<i>navStar-4-2-4</i>	t/out (p)	fail (m)			80% 359805 131 31		
<i>navStar-4-2-5</i>	100% 33235 9	19377 2474 20			100% 390803 192 28		
<i>navStar-4-2-6</i>	100% 43377 9	23057 4548 30			t/out (p)		
<i>navStar-4-2-7</i>	t/out (p)	fail (m)			80% 298879 124 37		
<i>navStar-4-2-8</i>	t/out (p)	fail (m)			t/out (p)		
<i>navStar-4-2-9</i>	t/out (p)	fail (m)			t/out (p)		
<i>navStar-4-2-10</i>	t/out (p)	fail (m)			t/out (p)		
<i>navStar-4-3-1</i>	100% 11395 7	40104 2178 17	100% 19550 1733 12		100% 36974 45 0	100% 64621 12 0	
<i>navStar-4-3-2</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-4-3-3</i>	100% 19770 7	13199 2196 16	100% 19381 2203 15		100% 53078 24 0	80% 96707 19 0	
<i>navStar-4-3-4</i>	100% 386381 73	192698 47090 175	fail (m)		t/out (p)	t/out (p)	
<i>navStar-4-3-5</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-4-3-6</i>	t/out (p)	fail (p)	fail (m)		80% 527918 447 36	20% 564518 83 22	
<i>navStar-4-3-7</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-4-3-8</i>	100% 48835 12	41952 2602 18	100% 17190 2038 13		100% 270059 182 0	20% 198028 34 0	
<i>navStar-4-3-9</i>	100% 170716 41	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-4-3-10</i>	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-5-2-1</i>	100% 302786 21	fail (m)			t/out (p)	t/out (p)	
<i>navStar-5-2-2</i>	100% 26801 6	32185 1757 16			t/out (p)	t/out (p)	
<i>navStar-5-2-3</i>	t/out (p)	fail (m)			t/out (p)	t/out (p)	
<i>navStar-5-2-4</i>	t/out (p)	fail (m)			t/out (p)	t/out (p)	
<i>navStar-5-2-5</i>	100% 366400 44	fail (m)			t/out (p)	t/out (p)	
<i>navStar-5-2-6</i>	t/out (p)	fail (m)			t/out (p)	t/out (p)	
<i>navStar-5-2-7</i>	100% 26419 5	33869 1139 12			t/out (p)	t/out (p)	
<i>navStar-5-2-8</i>	100% 84023 9	43413 3530 22			t/out (p)	t/out (p)	
<i>navStar-5-2-9</i>	100% 85030 9	44697 4639 24			t/out (p)	t/out (p)	
<i>navStar-5-2-10</i>	100% 247315 16	36720 3823 24			t/out (p)	t/out (p)	
<i>navStar-5-3-1</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-5-3-2</i>	100% 95264 11	80201 2720 18	100% 50554 5701 22		t/out (p)	20% 304508 10 0	
<i>navStar-5-3-3</i>	t/out (p)	fail (m)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-5-3-4</i>	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-5-3-5</i>	t/out (p)	fail (p)	fail (m)		t/out (p)	t/out (p)	
<i>navStar-5-3-6</i>	80% 365514 19	384401 15162 52	100% 51361 6430 25		t/out (p)	t/out (p)	
<i>navStar-5-3-7</i>	100% 132781 12	62455 6052 26	100% 53566 9802 28		t/out (p)	t/out (p)	

Problem	c	m2	m3	m4	d2	d3	d4
<i>navStar-5-3-8</i>	100% 232181	17 100%	40063 3144 21	100% 56527 4255 20	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-3-9</i>	100% 278276	20	fail (m)	100% 63090 13484 32	100% 187274 28 0	t/out (p)	t/out (p)
<i>navStar-5-3-10</i>	100% 417474	22	fail (p)	100% 55891 16682 41	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-1</i>	t/out (p)		fail (m)	t/out (p)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-2</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-3</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-4</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-5</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-6</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-7</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-8</i>	t/out (p)		t/out (p)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-9</i>	t/out (p)		fail (m)	fail (m)	t/out (p)	t/out (p)	t/out (p)
<i>navStar-5-4-10</i>	t/out (p)		t/out (p)	t/out (p)	t/out (p)	t/out (p)	t/out (p)

Table C.7: Complete experimental data for *holesGeneral* problems.

Problem	c	m3	d3	Problem	c	m3	d3
<i>holesGeneral-3-1-3-1</i>	100% 317 1	100% 1216 658 3	100% 31089 1 0	<i>holesGeneral-4-1-3-1</i>	100% 382 1	100% 1568 1262 4	t/out (p)
<i>holesGeneral-3-1-3-2</i>	100% 328 1	100% 1108 755 4	100% 36529 2 0	<i>holesGeneral-4-1-3-2</i>	100% 845 3	100% 1986 1794 4	t/out (p)
<i>holesGeneral-3-1-3-3</i>	100% 491 2	100% 1236 973 3	40% 71486 3 0	<i>holesGeneral-4-1-3-3</i>	100% 6555 14	100% 1776 1748 5	t/out (p)
<i>holesGeneral-3-1-3-4</i>	100% 694 3	100% 2944 2388 8	100% 52955 4 0	<i>holesGeneral-4-1-3-4</i>	100% 440 1	100% 1702 1164 4	t/out (p)
<i>holesGeneral-3-1-3-5</i>	100% 324 1	100% 1564 964 4	60% 46511 2 0	<i>holesGeneral-4-1-3-5</i>	100% 629 2	100% 1915 1452 4	t/out (p)
<i>holesGeneral-3-1-3-6</i>	100% 1254 7	100% 1406 941 3	40% 137614 2 0	<i>holesGeneral-4-1-3-6</i>	100% 1470 4	100% 1976 1779 5	t/out (p)
<i>holesGeneral-3-1-3-7</i>	100% 550 3	100% 1494 1102 3	80% 47506 3 0	<i>holesGeneral-4-1-3-7</i>	100% 572 2	100% 1784 1472 4	t/out (p)
<i>holesGeneral-3-1-3-8</i>	100% 380 2	100% 1353 924 3	34079 3 0	<i>holesGeneral-4-1-3-8</i>	100% 58 0	100% 1601 916 4	t/out (p)
<i>holesGeneral-3-1-3-9</i>	100% 352 1	40% 868 424 3	100% 25235 2 0	<i>holesGeneral-4-1-3-9</i>	100% 1120 3	100% 1576 1546 5	t/out (p)
<i>holesGeneral-3-1-3-10</i>	100% 407 2	100% 1734 1022 3	100% 104790 3 0	<i>holesGeneral-4-1-3-10</i>	100% 1138 4	100% 2386 1908 4	t/out (p)
<i>holesGeneral-3-1-4-1</i>	100% 287 1	100% 1132 760 4	100% 49929 1 0	<i>holesGeneral-4-1-4-1</i>	100% 434 1	100% 1808 1450 4	100% 508286 1 0
<i>holesGeneral-3-1-4-2</i>	100% 473 2	100% 1346 934 3	80% 36960 2 0	<i>holesGeneral-4-1-4-2</i>	100% 822 2	100% 2176 1504 4	100% 343902 2 0
<i>holesGeneral-3-1-4-3</i>	100% 492 2	100% 1685 1147 3	60% 55381 3 0	<i>holesGeneral-4-1-4-3</i>	100% 640 2	100% 2297 1764 4	100% 286867 1 0
<i>holesGeneral-3-1-4-4</i>	100% 412 1	100% 1342 827 3	100% 37004 1 0	<i>holesGeneral-4-1-4-4</i>	100% 592 2	100% 2054 1350 4	100% 402731 2 0
<i>holesGeneral-3-1-4-5</i>	100% 303 1	100% 1232 530 3	80% 61802 3 0	<i>holesGeneral-4-1-4-5</i>	100% 628 2	100% 2243 1623 4	100% 403393 2 0
<i>holesGeneral-3-1-4-6</i>	100% 475 2	100% 1606 1080 3	80% 25672 2 0	<i>holesGeneral-4-1-4-6</i>	100% 405 1	100% 1471 952 4	100% 254567 1 0
<i>holesGeneral-3-1-4-7</i>	100% 387 1	100% 1022 554 3	80% 52977 1 0	<i>holesGeneral-4-1-4-7</i>	100% 3076 7	100% 2178 1436 4	40% 292847 1 0
<i>holesGeneral-3-1-4-8</i>	100% 1293 7	100% 1845 1040 3	80% 40991 2 0	<i>holesGeneral-4-1-4-8</i>	100% 8940 16	100% 6650 5926 13	t/out (p)
<i>holesGeneral-3-1-4-9</i>	100% 1481 7	100% 1469 1004 3	40% 69050 3 0	<i>holesGeneral-4-1-4-9</i>	100% 926 2	100% 10375 9925 14	100% 276101 1 0
<i>holesGeneral-3-1-4-10</i>	100% 498 2	100% 1280 802 3	60% 70908 1 0	<i>holesGeneral-4-1-4-10</i>	100% 547 2	100% 1528 1268 4	100% 416058 3 0
<i>holesGeneral-3-2-3-1</i>	100% 1395 7	100% 1571 1179 3	100% 125001 3 0	<i>holesGeneral-4-2-3-1</i>	100% 574 2	100% 1247 886 5	100% 381054 1 0
<i>holesGeneral-3-2-3-2</i>	100% 51 0	100% 886 575 4	100% 91628 1 0	<i>holesGeneral-4-2-3-2</i>	100% 456 1	100% 1575 950 4	100% 434876 1 0
<i>holesGeneral-3-2-3-3</i>	100% 70 0	100% 1317 601 3	100% 104335 0 0	<i>holesGeneral-4-2-3-3</i>	100% 595 2	100% 1738 1372 4	100% 390329 1 0
<i>holesGeneral-3-2-3-4</i>	100% 231 1	100% 1186 684 3	100% 82578 1 0	<i>holesGeneral-4-2-3-4</i>	100% 416 1	100% 1541 1120 4	100% 408367 1 0
<i>holesGeneral-3-2-3-5</i>	100% 100 0	100% 1228 586 3	100% 98164 0 0	<i>holesGeneral-4-2-3-5</i>	100% 435 1	100% 1736 1073 5	100% 447140 1 0
<i>holesGeneral-3-2-3-6</i>	100% 74 0	100% 1056 727 3	100% 83394 1 0	<i>holesGeneral-4-2-3-6</i>	100% 62 0	100% 1571 1076 4	100% 338114 1 0
<i>holesGeneral-3-2-3-7</i>	100% 78 0	100% 774 466 3	100% 94970 0 0	<i>holesGeneral-4-2-3-7</i>	100% 644 2	100% 1776 1306 4	40% 585555 1 0
<i>holesGeneral-3-2-3-8</i>	100% 68 0	100% 1166 597 3	100% 105432 0 0	<i>holesGeneral-4-2-3-8</i>	100% 644 2	100% 1476 1080 5	100% 500859 1 0
<i>holesGeneral-3-2-3-9</i>	100% 74 0	100% 1079 606 3	100% 115258 0 0	<i>holesGeneral-4-2-3-9</i>	100% 397 1	100% 1416 1070 4	100% 269065 1 0
<i>holesGeneral-3-2-3-10</i>	100% 48 0	100% 897 488 3	100% 58841 0 0	<i>holesGeneral-4-2-3-10</i>	100% 598 2	100% 1683 1394 4	100% 424088 1 0
<i>holesGeneral-3-2-4-1</i>	100% 211 1	100% 1220 798 3	100% 65814 1 0	<i>holesGeneral-4-2-4-1</i>	100% 842 3	100% 2301 1968 4	t/out (p)

Problem	c	m3	d3	Problem	c	m3	d3
holesGeneral_3_2_4_2	100%	1167	720 3	holesGeneral_4_2_4_2	100%	1360	926 4
holesGeneral_3_2_4_3	100%	100%	1073	holesGeneral_4_2_4_3	100%	1593	1049 4
holesGeneral_3_2_4_4	100%	298 1	100%	holesGeneral_4_2_4_4	100%	1347	988 4
holesGeneral_3_2_4_5	100%	68 0	100%	holesGeneral_4_2_4_5	100%	1612	1246 4
holesGeneral_3_2_4_6	100%	312 1	100%	holesGeneral_4_2_4_6	100%	1775	1289 5
holesGeneral_3_2_4_7	100%	44 0	100%	holesGeneral_4_2_4_7	100%	1454	1297 5
holesGeneral_3_2_4_8	100%	32 0	100%	holesGeneral_4_2_4_8	100%	1403	855 4
holesGeneral_3_2_4_9	100%	42 0	100%	holesGeneral_4_2_4_9	100%	1136	899 4
holesGeneral_3_2_4_10	100%	71 0	100%	holesGeneral_4_2_4_10	100%	1510	926 4
holesGeneral_3_3_3_1	100%	314 1	100%	holesGeneral_4_3_3_1	100%	1888	1367 4
holesGeneral_3_3_3_2	100%	432 2	100%	holesGeneral_4_3_3_2	100%	1449	850 4
holesGeneral_3_3_3_3	100%	290 1	100%	holesGeneral_4_3_3_3	100%	1766	1136 4
holesGeneral_3_3_3_4	100%	324 1	100%	holesGeneral_4_3_3_4	100%	1528	1042 5
holesGeneral_3_3_3_5	100%	306 1	100%	holesGeneral_4_3_3_5	100%	1388	898 4
holesGeneral_3_3_3_6	100%	54 0	100%	holesGeneral_4_3_3_6	100%	1406	936 4
holesGeneral_3_3_3_7	100%	316 1	100%	holesGeneral_4_3_3_7	100%	1440	995 4
holesGeneral_3_3_3_8	100%	292 1	100%	holesGeneral_4_3_3_8	100%	1280	832 4
holesGeneral_3_3_3_9	100%	55 0	100%	holesGeneral_4_3_3_9	100%	2150	1714 4
holesGeneral_3_3_3_10	100%	414 2	100%	holesGeneral_4_3_3_10	100%	1569	1144 5
holesGeneral_3_3_4_1	100%	317 1	100%	holesGeneral_4_3_4_1	100%	1335	841 4
holesGeneral_3_3_4_2	100%	313 1	100%	holesGeneral_4_3_4_2	100%	1317	862 4
holesGeneral_3_3_4_3	100%	312 1	100%	holesGeneral_4_3_4_3	100%	t/out (m)	100%
holesGeneral_3_3_4_4	100%	297 1	100%	holesGeneral_4_3_4_4	100%	1326	905 4
holesGeneral_3_3_4_5	100%	320 1	100%	holesGeneral_4_3_4_5	100%	1479	934 4
holesGeneral_3_3_4_6	100%	311 1	100%	holesGeneral_4_3_4_6	100%	1778	1074 5
holesGeneral_3_3_4_7	100%	339 1	100%	holesGeneral_4_3_4_7	100%	t/out (m)	100%
holesGeneral_3_3_4_8	100%	291 1	100%	holesGeneral_4_3_4_8	100%	1378	858 4
holesGeneral_3_3_4_9	100%	301 1	100%	holesGeneral_4_3_4_9	100%	1365	931 4
holesGeneral_3_3_4_10	100%	292 1	100%	holesGeneral_4_3_4_10	100%	2928	1975 9
holesSpecial_3_1_3_1	100%	26369 18	100%	holesSpecial_4_1_3_1	100%	162539 24	100%
holesSpecial_3_1_3_2	100%	26135 20	100%	holesSpecial_4_1_3_2	100%	148704 28	100%
holesSpecial_3_1_3_3	100%	29200 21	100%	holesSpecial_4_1_3_3	100%	125578 24	100%
holesSpecial_3_1_3_4	100%	25414 18	100%	holesSpecial_4_1_3_4	100%	173486 62	100%
holesSpecial_3_1_3_5	100%	28009 21	100%	holesSpecial_4_1_3_5	t/out (p)	100%	384671 3606 31
holesSpecial_3_1_3_6	100%	28860 21	100%	holesSpecial_4_1_3_6	100%	149712 24	100%
holesSpecial_3_1_3_7	100%	24220 20	100%	holesSpecial_4_1_3_7	100%	129265 28	100%
holesSpecial_3_1_3_8	100%	28405 47	100%	holesSpecial_4_1_3_8	100%	146876 38	100%
holesSpecial_3_1_3_9	100%	24063 26	100%	holesSpecial_4_1_3_9	t/out (p)	100%	455550 4213 21
holesSpecial_3_1_3_10	100%	32697 18	100%	holesSpecial_4_1_3_10	100%	127009 24	100%
holesSpecial_3_1_4_1	100%	35309 21	100%	holesSpecial_4_1_4_1	100%	208200 24	100%
holesSpecial_3_1_4_2	100%	33164 20	100%	holesSpecial_4_1_4_2	100%	162455 24	100%
holesSpecial_3_1_4_3	100%	33044 30	100%	holesSpecial_4_1_4_3	100%	174717 28	100%
holesSpecial_3_1_4_4	100%	33698 21	100%	holesSpecial_4_1_4_4	100%	167342 28	100%
holesSpecial_3_1_4_5	100%	29740 21	100%	holesSpecial_4_1_4_5	100%	188606 51	100%
holesSpecial_3_1_3_1	20%	433049 24	5	holesSpecial_4_1_3_1	100%	251061 3934 21	t/out (p)
holesSpecial_3_1_3_2	t/out (p)	62281 1962 16	16	holesSpecial_4_1_3_2	100%	300334 3892 32	t/out (p)
holesSpecial_3_1_3_3	t/out (p)	63788 1999 16	16	holesSpecial_4_1_3_3	100%	186381 3632 24	t/out (p)
holesSpecial_3_1_3_4	t/out (p)	85639 2184 16	16	holesSpecial_4_1_3_4	100%	371594 3544 34	t/out (p)
holesSpecial_3_1_3_5	20%	64358 1907 16	4	holesSpecial_4_1_3_5	t/out (p)	100%	384671 3606 31
holesSpecial_3_1_3_6	20%	73506 1960 15	4	holesSpecial_4_1_3_6	100%	186351 3686 30	t/out (p)
holesSpecial_3_1_3_7	t/out (p)	85697 2237 16	16	holesSpecial_4_1_3_7	100%	189534 3631 21	t/out (p)
holesSpecial_3_1_3_8	t/out (p)	56457 1863 16	16	holesSpecial_4_1_3_8	100%	269078 3922 32	t/out (p)
holesSpecial_3_1_3_9	20%	41434 1890 16	20	holesSpecial_4_1_3_9	100%	455550 4213 21	t/out (p)
holesSpecial_3_1_3_10	20%	51710 1848 16	23	holesSpecial_4_1_3_10	100%	253072 3669 35	t/out (p)
holesSpecial_3_1_4_1	t/out (p)	121052 2141 16	16	holesSpecial_4_1_4_1	100%	119476 4369 27	t/out (p)
holesSpecial_3_1_4_2	t/out (p)	72796 2258 16	16	holesSpecial_4_1_4_2	100%	622788 3720 35	t/out (p)
holesSpecial_3_1_4_3	t/out (p)	62344 2234 16	16	holesSpecial_4_1_4_3	100%	174717 28	100%
holesSpecial_3_1_4_4	t/out (p)	48647 1949 16	16	holesSpecial_4_1_4_4	100%	56075 3700 21	t/out (p)
holesSpecial_3_1_4_5	t/out (p)	64170 2249 15	15	holesSpecial_4_1_4_5	100%	66944 3562 41	t/out (p)

Table C.8: Complete experimental data for holesSpecial problems.

Problem	c	m3	d3	Problem	c	m3	d3
holesSpecial_3_1_4_6	100% 33657 21	100% 61377 2242 16	t/out (p)	holesSpecial_4_1_4_6	100% 180284	24 100% 71593 4092 25	t/out (p)
holesSpecial_3_1_4_7	100% 39333 30	100% 72094 2279 16	t/out (p)	holesSpecial_4_1_4_7	100% 171245	24 100% 66328 3762 27	t/out (p)
holesSpecial_3_1_4_8	100% 30232 18	100% 48518 1951 16	t/out (p)	holesSpecial_4_1_4_8	100% 267221	127 100% 59225 3605 36	t/out (p)
holesSpecial_3_1_4_9	100% 33426 18	100% 64109 2270 16	t/out (p)	holesSpecial_4_1_4_9	100% 179562	28 100% 69248 3773 30	t/out (p)
holesSpecial_3_1_4_10	100% 32409 18	100% 63054 2268 18	t/out (p)	holesSpecial_4_1_4_10	100% 162743	24 100% 59313 3994 24	t/out (p)
holesSpecial_3_2_3_1	100% 44938 45	100% 51080 1846 16	t/out (p)	holesSpecial_4_2_3_1	100% 136818	61 100% 170171 3470 34	t/out (p)
holesSpecial_3_2_3_2	100% 35088 20	100% 51778 1846 16	t/out (p)	holesSpecial_4_2_3_2	100% 126793	26 100% 216982 3700 25	t/out (p)
holesSpecial_3_2_3_3	100% 43320 30	100% 66147 1978 15	t/out (p)	holesSpecial_4_2_3_3	100% 141800	69 100% 194964 3680 27	t/out (p)
holesSpecial_3_2_3_4	100% 35362 20	100% 52231 1850 15	t/out (p)	holesSpecial_4_2_3_4	100% 157248	74 100% 190301 3667 27	t/out (p)
holesSpecial_3_2_3_5	100% 37444 21	100% 54543 1941 15	40% 651186 26 2	holesSpecial_4_2_3_5	100% 148218	62 100% 229927 3738 29	t/out (p)
holesSpecial_3_2_3_6	100% 36248 21	100% 51050 1896 15	t/out (p)	holesSpecial_4_2_3_6	100% 186310	115 100% 169750 3542 29	t/out (p)
holesSpecial_3_2_3_7	100% 36024 20	100% 51858 1925 15	t/out (p)	holesSpecial_4_2_3_7	100% 149659	52 100% 248837 3918 31	t/out (p)
holesSpecial_3_2_3_8	100% 41080 21	100% 62366 1974 15	t/out (p)	holesSpecial_4_2_3_8	100% 143645	69 100% 234080 3818 28	t/out (p)
holesSpecial_3_2_3_9	100% 38240 20	100% 66305 1937 17	t/out (p)	holesSpecial_4_2_3_9	100% 105078	24 100% 102062 3294 21	t/out (p)
holesSpecial_3_2_3_10	100% 32598 20	100% 38359 1868 16	t/out (p)	holesSpecial_4_2_3_10	100% 160895	77 100% 187419 3692 30	t/out (p)
holesSpecial_3_2_4_1	100% 42334 32	100% 34264 1950 17	t/out (p)	holesSpecial_4_2_4_1	100% 199120	103 100% 41062 3668 30	t/out (p)
holesSpecial_3_2_4_2	100% 46914 30	100% 46485 1921 16	t/out (p)	holesSpecial_4_2_4_2	100% 149748	39 100% 91778 3602 38	t/out (p)
holesSpecial_3_2_4_3	100% 43398 20	100% 48762 1966 17	t/out (p)	holesSpecial_4_2_4_3	100% 176025	62 100% 58226 3642 33	t/out (p)
holesSpecial_3_2_4_4	100% 41264 21	100% 33212 1972 16	t/out (p)	holesSpecial_4_2_4_4	100% 144578	28 100% 75976 3578 21	t/out (p)
holesSpecial_3_2_4_5	100% 41500 20	100% 37700 2148 15	40% 606990 25 4	holesSpecial_4_2_4_5	100% 153938	24 100% 49198 3684 21	t/out (p)
holesSpecial_3_2_4_6	100% 47328 30	100% 47838 1912 16	40% 526949 19 2	holesSpecial_4_2_4_6	100% 158959	54 100% 68954 3402 35	t/out (p)
holesSpecial_3_2_4_7	100% 36702 20	100% 30624 1859 16	t/out (p)	holesSpecial_4_2_4_7	100% 157038	50 100% 67941 3534 27	t/out (p)
holesSpecial_3_2_4_8	100% 41620 21	100% 37208 1964 18	t/out (p)	holesSpecial_4_2_4_8	100% 174202	75 100% 49989 3696 25	t/out (p)
holesSpecial_3_2_4_9	100% 41274 32	100% 31590 1998 18	t/out (p)	holesSpecial_4_2_4_9	100% 204794	75 100% 49104 3674 32	t/out (p)
holesSpecial_3_2_4_10	100% 49012 30	100% 48700 2195 15	t/out (p)	holesSpecial_4_2_4_10	100% 185748	75 100% 79208 3678 29	t/out (p)
holesSpecial_3_3_3_1	100% 20165 18	100% 33146 1874 15	100% 512793 55 12	holesSpecial_4_3_3_1	100% 70133	28 100% 110266 3162 21	t/out (p)
holesSpecial_3_3_3_2	100% 20546 19	100% 33838 1929 15	60% 576700 66 12	holesSpecial_4_3_3_2	100% 69316	27 100% 110033 2980 21	t/out (p)
holesSpecial_3_3_3_3	100% 20937 19	100% 33454 1930 15	20% 121312 11 1	holesSpecial_4_3_3_3	100% 68896	24 100% 112626 3080 20	t/out (p)
holesSpecial_3_3_3_4	100% 20564 20	100% 33250 1846 15	20% 167744 15 3	holesSpecial_4_3_3_4	100% 69520	24 100% 110464 3130 20	t/out (p)
holesSpecial_3_3_3_5	100% 20306 19	100% 32878 1846 15	40% 242038 26 5	holesSpecial_4_3_3_5	100% 69340	27 100% 110136 2996 19	t/out (p)
holesSpecial_3_3_3_6	100% 20365 18	100% 33156 1904 15	80% 310840 32 8	holesSpecial_4_3_3_6	100% 68446	27 100% 111082 3043 20	t/out (p)
holesSpecial_3_3_3_7	100% 20297 18	100% 32918 1900 15	20% 192004 20 4	holesSpecial_4_3_3_7	100% 69693	26 100% 111680 3059 21	t/out (p)
holesSpecial_3_3_3_8	100% 20507 19	100% 33584 1850 15	80% 406902 44 8	holesSpecial_4_3_3_8	100% 69483	26 100% 112435 3258 20	t/out (p)
holesSpecial_3_3_3_9	100% 20445 19	100% 33129 1812 15	100% 213597 20 1	holesSpecial_4_3_3_9	100% 71908	27 100% 110128 3006 21	t/out (p)
holesSpecial_3_3_3_10	100% 20845 19	100% 33435 1933 15	20% 911165 8 0	holesSpecial_4_3_3_10	100% 69689	27 100% 111264 3058 20	t/out (p)
holesSpecial_3_3_4_1	100% 23767 19	100% 26635 1886 15	80% 349316 43 5	holesSpecial_4_3_4_1	100% 89797	26 100% 45856 2826 21	t/out (p)
holesSpecial_3_3_4_2	100% 23958 18	100% 26802 1871 15	60% 295644 36 6	holesSpecial_4_3_4_2	100% 90611	26 100% 46954 3360 20	t/out (p)
holesSpecial_3_3_4_3	100% 23832 19	100% 26564 1888 15	60% 384941 51 9	holesSpecial_4_3_4_3	100% 90659	26 100% 47662 3418 19	t/out (p)
holesSpecial_3_3_4_4	100% 23681 19	100% 26464 1888 14	100% 298568 40 8	holesSpecial_4_3_4_4	100% 89927	27 100% 45846 3320 20	t/out (p)
holesSpecial_3_3_4_5	100% 23807 19	100% 26602 1898 15	20% 366923 48 10	holesSpecial_4_3_4_5	100% 91421	26 100% 47202 3266 20	20% 548170 18 0
holesSpecial_3_3_4_6	100% 23739 19	100% 26291 1908 15	80% 324911 42 7	holesSpecial_4_3_4_6	100% 90263	28 100% 42261 3386 20	t/out (p)
holesSpecial_3_3_4_7	100% 23741 19	100% 25753 1898 15	t/out (p)	holesSpecial_4_3_4_7	100% 91695	27 100% 43804 3378 20	t/out (p)
holesSpecial_3_3_4_8	100% 23992 18	100% 26454 1888 15	100% 145397 15 0	holesSpecial_4_3_4_8	100% 90687	26 100% 46987 3336 19	t/out (p)
holesSpecial_3_3_4_9	100% 23912 19	100% 26120 1897 14	100% 363771 48 8	holesSpecial_4_3_4_9	100% 89463	25 100% 46888 3268 20	t/out (p)
holesSpecial_3_3_4_10	100% 23640 18	100% 26264 1879 14	100% 208453 25 3	holesSpecial_4_3_4_10	100% 88834	25 100% 93103 6800 39	40% 430743 16 2

Bibliography

Rachid Alami, Sara Fleury, Matthieu Herrb, Félix Ingrand, and Frédéric Robert. Multi robot cooperation in the Martha Project. IEEE Robotics and Automation Magazine: special issue entitled “Robotics and Automation in Europe: Projects funded by the Commission of the European Union”, 1997. URL <ftp://ftp.laas.fr/pub/Publications/1996/96392.ps>.

Rachid Alami, Félix Ingrand, and Samer Qutub. A scheme for coordinating multi-robot planning activities and plans execution. In Proceedings of the Thirteenth European Conference on Artificial Intelligence, Brighton, UK, 1998. URL <ftp://ftp.laas.fr/pub/ria/felix/publis/ecai98.ps.gz>.

Marco Alberti, Marco Gavanelli, Evelina Lamma, Federico Chesani, Paola Mello, and Paolo Torroni. A logic based approach to interaction design in open multi-agent systems, September 2004. URL <http://citeseer.ist.psu.edu/alberti04logic.html>.

James Allen, James Hendler, and Austin Tate, editors. Readings in Planning. Morgan Kaufman, 1990.

James F. Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, 26(11):832–843, 1983.

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. Artificial Intelligence, 116:123–191, 2000. URL <http://www.cs.toronto.edu/~fbacchus/tlplan.html>.

Luc Beaudoin. Goal Selection for Autonomous Agents. PhD thesis, University of Birming-

-
- ham, Edgbaston, Birmingham, UK, August 1994. URL http://www.cs.bham.ac.uk/research/cogaff/Luc.Beaudoin_thesis.pdf.
- Keith Biggers and Thomas Ioerger. Automatic generation of communication and teamwork within multi-agent teams. *Applied Artificial Intelligence*, 15:875–916, 2001.
- Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997. URL <http://citeseer.nj.nec.com/blum95fast.html>.
- Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1988.
- Blai Bonet and Hector Geffner. HSP: Heuristic Search Planner. In *Proceedings of the Planning Competition at the Fourth International Conference on Artificial Intelligence Planning Systems Planning Competition*, 1998. URL <http://citeseer.ist.psu.edu/137014.html>.
- Ronen I. Brafman and Holger H. Hoos. To encode or not to encode - i: Linear planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.
- Michael Brenner. Multiagent planning with partially ordered temporal plans. In *Proceedings of the Doctorial Consortium of the International Conference on AI Planning and Scheduling*, 2003. URL http://icaps03.itc.it/satellite_events/doctoral_consortium.htm.
- Will Briggs and Diane J. Cook. *Modularity and Communication in Multi-Agent Planning*. PhD thesis, University of Texas at Arlington, 1996. URL <http://citeseer.nj.nec.com/briggs96modularity.html>.
- Rodney A. Brooks. How to build complete creatures rather than isolated cognitive simulators. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 225–239. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991. URL <http://ai.eecs.umich.edu/cogarch3/Brooks/Brooks.html>.

- Michael Browning, Brett Browning, and Manuela Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling, Vancouver, June 2004. URL <http://www-2.cs.cmu.edu/~mmv/pubs04.html>.
- David Chapman. Planning for conjunctive goals. Artificial Intelligence, 32:333–377, 1987. Reprinted in Allen et al. (1990).
- Steve Chien, Russell Knight, André Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to increase the responsiveness of planning and scheduling for autonomous spacecraft. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence Workshop: Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World, 1999.
- Bradley J. Clement. Abstract Reasoning for Multiagent Coordination and Planning. PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, 2002. URL <http://www-personal.umich.edu/~bradc/papers/clement-diss.pdf>.
- Bradley J. Clement and Edmund H. Durfee. Identifying and resolving conflicts among agents with hierarchical plans. In AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, number WS-99-12 in AAI Technical Report, pages 6–11, 1999a.
- Bradley J. Clement and Edmund H. Durfee. Theory for coordinating concurrent hierarchical planning agents using summary information. In Proceedings of the Sixteenth National Conference on Artificial Intelligence, pages 495–502, 1999b.
- Bradley J. Clement and Edmund H. Durfee. Top-down search for coordinating the hierarchical plans or multiple agents. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, Proceedings of the Third International Conference on Autonomous Agents (Agents'99), pages 252–259, Seattle, WA, USA, 1999c. ACM Press. URL <http://citeseer.nj.nec.com/clement99topdown.html>.
- Mathijs M. de Weerd. Plan Merging in Multi-Agent Systems. PhD thesis, Delft Technical University, Delft, The Netherlands, 2003. URL www.pds.twi.tudelft.nl/~mathijs/.

- Mathijs M. de Weerd and Roman P.J. van der Krogt. A method to integrate planning and coordination. In Michael Brenner and Marie desJardins, editors, Planning with and for Multiagent Systems, number WS-02-12 in AAI Technical Report, pages 83–88, Menlo Park, CA, 2002. AAI Press.
- Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. Artif. Intell., 49(1-3): 61–95, 1991. ISSN 0004-3702. doi: [http://dx.doi.org/10.1016/0004-3702\(91\)90006-6](http://dx.doi.org/10.1016/0004-3702(91)90006-6).
- Keith S. Decker and Victor R. Lesser. Generalizing the Partial Global Planning algorithm. International Journal of Intelligent and Cooperative Information Systems, 1(2):319–346, 1992. URL <http://citeseer.nj.nec.com/decker92generalizing.html>.
- Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. Technical Report 94-I4, Department of Computer Science, University of Massachusetts, 1995. URL <http://citeseer.nj.nec.com/decker95designing.html>.
- Marie E. desJardins, Edmund H. Durfee, Charles L. Ortiz Jr., and Michael J. Wolverton. A survey of research in Distributed, Continual Planning. AI Magazine, 2000.
- Minh B. Do and Subbarao Kambhampati. SAPA: A domain-independent heuristic metric temporal planner. In Proceedings of the Sixth European Conference on Planning (ECP-01), pages 109–120, 2001.
- Jim E. Doran, Stan Franklin, Nicholas R. Jennings, and Tim J. Norman. On cooperation in Multi-Agent Systems. The Knowledge Engineering Review, 12(3), 1997. URL <http://www.csd.abdn.ac.uk/~tnorman/publications/fomas.html>.
- Edmund H. Durfee and Victor R. Lesser. Partial Global Planning: A coordination framework for distributed hypothesis formation. IEEE Transactions on Systems Man and Cybernetics, 21(1):63–83, 1991. URL <http://citeseer.nj.nec.com/durfee91partial.html>.
- Eithan Ephrati and Jeffrey S. Rosenschein. Divide and conquer in multi-agent planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), pages 375–380, Menlo Park, CA, 1994. AAI Press.

- Kutlahan Erol. Hierarchical Task Network planning: formalization, analysis, and implementation. PhD thesis, Computer Science Department, University of Maryland, 1996. URL http://techreports.isr.umd.edu/TechReports/ISR/1996/PhD_96-4/PhD_96-4.pdf.
- Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 5(2):189–208, 1971.
- Maria Fox and Derek Long. Pddl2.1: An extension of pddl for expressing temporal planning domains. Journal of AI Research, 20:61–124, 2003. URL <http://www.jair.org>.
- Jeremy Frank, Ari K. Jónsson, and Paul Morris. On reformulating planning as dynamic constraint satisfaction. In Proceedings of the Symposium on Abstraction, Reformulation and Abstraction. Springer Verlag, July 2000. URL <http://ic.arc.nasa.gov/people/jonsson/Papers/sara2000.ps>.
- Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages. Springer-Verlag, 1996. URL <http://www.msci.memphis.edu/~franklin/AgentProg.html>.
- Michael P. Georgeff. Communication and interaction in multi-agent planning. In Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83), pages 125–129, Menlo Park, CA, 1983. AAAI Press. Reprinted in (Bond and Gasser, 1988).
- Malik Ghallab, Dana Nau, and Paolo Traverso. Automated Planning: Theory and Practice. Morgan Kaufmann, 2004.
- Keith Golden and Daniel S. Weld. Representing sensing actions: The middle ground revisited. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR '96), pages 174–185. Morgan Kaufmann, San Francisco, California, 1996. URL <http://citeseer.ist.psu.edu/article/golden96representing.html>.

- Elizabeth Gordon and Brian Logan. A goal processing architecture for game agents. Technical Report NOTTCS-WP-2003-1, School of Computer Science and Information Technology, University of Nottingham, 2002.
- Dave Gurnell. Adaptive coordination for multi agent planning. In Proceedings of the Twenty Second Workshop of the UK Planning and Scheduling Special Interest Group, 2003.
- Dave Gurnell. Distributed planning with summary information in recursive HTN domains. In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling. AAAI Press, June 2004.
- Nick Hawes. Anytime Deliberation for Computer Game Agents. PhD thesis, University of Birmingham, November 2003.
- Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. Journal of AI Research, 14:253–302, 2001. URL <http://www.informatik.uni-freiburg.de/~hoffmann/publications.html>.
- Tad Hogg, Bernardo A. Huberman, and Colin Williams. Phase transitions and the search problem. Artificial Intelligence: special issue entitled “Frontiers in Problem Solving: Phase Transitions and Complexity”, 81(1-2):1–15, March 1996. URL <http://www.hpl.hp.com/research/idl/projects/constraints/specialAIJ/specialAIJ.html>.
- Bryan Horling, Victor Lesser, Regis Vincent, Tom Wagner, Anita Raja, Shelley Zhang, Keith Decker, and Alan Garvey. The TAEMS white paper, January 1999. URL <http://mas.cs.umass.edu/paper/182>.
- Nicholas R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. The Knowledge Engineering Review, 8(3):223–250, 1993. URL <http://citeseer.nj.nec.com/jennings93commitments.html>.
- Subbarao Kambhampati. Refinement planning as a unifying framework for plan synthesis. AI Magazine, 18(2):67–97, 1997. URL <http://rakaposhi.eas.asu.edu/papers.html>.

- Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, Proceedings of the Tenth European Conference on Artificial Intelligence, pages 359–363, 1992. URL <http://www.cs.washington.edu/homes/kautz/papers/satplan.ps>.
- Richard Korf. Planning as search: A quantitative approach. Artificial Intelligence, 33(1):65–88, 1987. Republished in Allen et al. (1990).
- David Marr David Marr. Vision. Freeman Publishers, 1982.
- David McAllester and David Rosenblitt. Systematic nonlinear planning. In Proceedings of the Ninth National Conference on Artificial Intelligence, volume 2, pages 634–639, Anaheim, CA, 1991. AAAI Press/MIT Press. ISBN 0-262-51059-6. URL <http://citeseer.ist.psu.edu/mcallester91systematic.html>.
- T. Lee McCluskey, Donghong Liu, and Ron M. Simpson. Using knowledge engineering and state space planning techniques to optimise an HTN planner. In Proceedings of the Twenty First Workshop of the UK Planning and Scheduling Special Interest Group, 2002. URL <http://pds.twi.tudelft.nl/activities/PLANSIG2002/Programme.htm>.
- Drew McDermott. PDDL – the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998. URL <http://www.cs.yale.edu/homes/dvm/>.
- Erica Melis and Andreas Meier. Proof planning with multiple strategies. In Proceedings of the First International Conference on Computational Logic, London, 2000. URL <http://www.ags.uni-sb.de/~ameier/publications/2000/cl00.ps.gz>.
- Alexander Narayek. EXCALIBUR: Adaptive constraint-based agents in artificial environments: Online documentation. Web page, 2002. URL <http://www.ai-center.com/projects/excalibur/documentation/>.
- Dana Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz Avila. SHOP: A Simple Hierarchical Ordered Planner. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999.

- Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. Journal of Artificial Intelligence Research, 20: 379–404, March 2003.
- Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In Edward A. Feigenbaum and Julian Feldman, editors, Computers and Thought, pages 279–293, 1969. Reprinted in Allen et al. (1990).
- XuanLong Nguyen, Subbarao Kambhampati, and Romeo Sanchez Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search. Artificial Intelligence, March 2002.
- Nils J. Nilsson. Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research, 1:139–158, 1994.
- Edwin P. D. Pednault. Formulating multi-agent dynamic-world problems in the classical planning framework. In Michael P. Georgeff and Amy L. Lansky, editors, Reasoning About Actions and Plans: Proceedings of the 1986 Workshop, pages 47–82, San Mateo, CA, 1987. Morgan Kaufmann Publishers. Reprinted in Allen et al. (1990).
- J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR&R-92), pages 103–114. Morgan Kaufmann Publishers, October 1992.
- J. Scott Penberthy and Daniel S. Weld. Temporal planning with continuous change. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), pages 1010–1015, Menlo Park, CA, 1994. AAAI Press. URL <http://www.cs.washington.edu/homes/weld/pubs.html>.
- Gregg Rabideau, Russell Knight, Steve Chien, Alex Fukunaga, and Anita Govindjee. Iterative repair planning for spacecraft operations in the ASPEN system. In Proceedings of the Fifth International Symposium on Artificial Intelligence Robotics and Automation in Space, pages 99–106, Noordwijk, the Netherlands, 1999. ESA Publications Division.
- Craig W. Reynolds. Steering behaviours for autonomous characters. In Proceedings of the

-
- Game Developers Conference 1999, pages 763–782, San Francisco, California, 1999. Miller Freeman Game Group.
- Jussi Rintanen. Phase transitions in classical planning: an experimental study. In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling. AAAI Press, June 2004.
- Jeffrey S. Rosenschein. Synchronization of multi-agent plans. In Proceedings of the Second National Conference on Artificial Intelligence (AAAI-82), pages 115–119, Menlo Park, CA, 1982. AAAI Press. Reprinted in (Bond and Gasser, 1988).
- Jeffrey S. Rosenschein and Gilad Zlotkin. Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers. MIT Press, 1994.
- Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 1995. ISBN 0-13-360124-2.
- Earl D. Sacerdoti. The nonlinear nature of plans. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, pages 206–214, 1975.
- Eddie Schwalb and Lluís Vila. Temporal constraints: A survey. Constraints: An International Journal, 3(2/3):129–149, June 1998. URL <http://www.lsi.upc.es/~vila/vila/tcs/tcs.ps.gz>.
- Yoav Shoham and Moshe Tennenholtz. On social laws for artificial agent societies: Off-line design. Artificial Intelligence, 73(1–2):231–252, 1995. URL <http://citeseer.nj.nec.com/shoham95social.html>.
- Aaron Sloman. Interacting trajectories in design space and niche space: A philosopher speculates about evolution. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyn Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, Parallel Problem Solving from Nature — PPSN VI 6th International Conference, number 1917 in Lecture Notes in Computer Science, pages 3–16, Paris, France, September 2000.
- Aaron Sloman. Beyond shallow models of emotion. Cognitive Processing, 2(1):178–198, 2001. URL <http://www.cs.bham.ac.uk/research/cogaff/sloman.iqcs01.pdf>.

- Aaron Sloman. Architecture-based conceptions of mind. In Peter Gärdenfors, Katarzyna Kijania-Placek, and Jan Woleński, editors, In the Scope of Logic, Methodology, and Philosophy of Science (Vol II), volume 316 of Synthese Library, pages 403–427. Springer, 2002.
- David E. Smith and Daniel S. Weld. Temporal planning with mutual exclusion reasoning. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), San Mateo, CA, 1999. Morgan Kaufmann Publishers. URL <http://ic.arc.nasa.gov/ic/people/de2smith/publications/publications.html>.
- Reid G. Smith. The Contract Net Protocol: High level communication and control in a distributed problem solver. IEEE Transactions on Computers, C-29(2):1104–1113, 1980.
- Biplav Srivastava, Subbarao Kambhampati, and Minh B. Do. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in RealPlan. Artificial Intelligence, 131(1-2):73–134, September 2001.
- Austin Tate. Generating project networks. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, pages 888–893, 1977.
- Ioannis Tsamardinos, Martha E. Pollack, and John F. Horty. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-00), pages 264–272, Menlo Park, CA, 2000. AAAI Press.
- Reiko Tsuneto, James A. Hendler, and Dana S. Nau. Analyzing external conditions to improve the efficiency of HTN planning. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Menlo Park, CA, 1998. AAAI Press. URL <http://www.cs.umd.edu/users/reiko>.
- Jeroen M. Valk, Mathijs M. de Weerd, and Cees Witteveen. Algorithms for coordination in multi-agent planning. In Ioannis Vlahavas and Dimitris Vrakas, editors, Intelligent Techniques for Planning, pages 194–224, London, 2005. Idea Group Publishing.
- Daniel S. Weld. An introduction to least-commitment planning. AI Magazine, 15(4):27–61,

- Winter 1994. URL <http://citeseer.ist.psu.edu/weld94introduction.html>.
- Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, Summer 1999. URL <http://www.cs.washington.edu/homes/weld/weld.html>.
- Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98), 1998.
- David E. Wilkins and Karen L. Myers. A Multiagent Planning Architecture. In Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98), pages 154–162, Menlo Park, CA, 1998. AAAI Press. URL <http://www.ai.sri.com/~wilkins/bib.html>. Also available as a technical report.
- Michael Wooldridge. Reasoning about Rational Agents. Intelligent robotics and autonomous agents. MIT Press, Cambridge, Massachusetts/London, England, June 2000. URL <http://www.csc.liv.ac.uk/~mjw/pubs/rara/>.
- Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1994. URL <http://citeseer.ist.psu.edu/article/wooldridge95intelligent.html>. Revised and republished on the Web in 1995.
- Qiang Yang. Intelligent Planning: A Decomposition and Abstraction Based Approach. Springer Verlag, New York, 1997. ISBN 3-540-61901-1.
- Qiang Yang, Dana S. Nau, and James Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(4):648–676, 1992. URL <http://www.cs.sfu.ca/~isa/pubs/index.html>.
- Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review of "plans and situated actions" by Lucy Suchman. Autonomous Agents and Multi-Agent Systems, 2000.
- Gilad Zlotkin and Jeffrey S. Rosenschein. Mechanisms for automated negotiation in state oriented domains. *Journal of Artificial Intelligence Research*, 5:163–238, 1996.