MANAGING TIME BUDGETS SHARED BETWEEN PLANNING AND EXECUTION

by

JACK ELLIOT HARGREAVES

A thesis submitted to the University of Birmingham for the degree of DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
September 2016

Abstract

Agents operating in domains with time budgets shared between planning and execution must carefully balance the need to plan versus the need to act. This is because planning and execution consume the same time resource. Excessive planning can delay the time it takes to achieve a goal, and so reduce the reward attained by an agent. Whereas, insufficient planning will mean the agent creates and executes low reward plans.

This thesis looks at three ways to increase the reward achieved by an agent in domains with shared time budgets. The first way is by optimising time allocated to planning, using two different methods – an optimal plan duration predictor and an online loss limiter. A second is by finding ways to act in a goal-directed manner during planning. We look at using previous plans or new plans generated quickly as heuristics for acting whilst planning. In addition, we present a way of describing actions that are mid-execution to speed the transition between planning and execution. Lastly, this thesis presents a way in which to manage time budgets in multi-agent domains. We use market-based task allocation with deadlines to produce faster task allocation and planning.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and inspiration of many people. First and foremost, I would like to thank my supervisor, Nick Hawes, whose encouragement and knowledge have enabled me to complete my research. I am very grateful for the advice and discussions we have had throughout my years at Birmingham.

I should also like to thank Jeremy Baxter and Rustam Stolkin who co-supervised my research in the early years of PhD. In addition, the pastoral care that Rustam offered has helped me immensely during tougher periods.

The tireless efforts of the administrative staff of the Computer Science department have been of huge help, and made my time at Birmingham that much easier. I would especially like to thank Julie Heathcote and Caroline Wilson.

To my friends, thank you for all the wonderful times and memories. For keeping me balanced when I was at risk of getting lost in my research.

Finally, I wish to express my undying gratitude to my Mum. Without whose constant support and encouragement I would not be where I am today.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The research question addressed by this thesis is how can a goal-oriented system share its time effectively between planning and plan execution. This is a fundamental problem in many real world domains since time continues to pass while agents are planning, and this passage of time can impact the usefulness of the results of the planning process. This question is addressed by decomposing it into three related sub-questions:

1. How long should an agent plan in the presence of a fixed time budget?
2. What should an agent do when planning? That is, is it better that an agent should wait until a plan is ready before acting, or should the agent act in the meantime.
3. In domains where timing is an important factor in behaviour, should planning aim to generate high quality plans, or generate plans quicker?

1.1.1 How Long Should an Agent Plan for?

The main problem when planning and execution have a shared time budget is that if the agent plans for too long then the resulting plan may not be executable within the remaining time budget. At its most extreme, what would have been the optimal plan at

the beginning of planning is not valid, much less optimal, if there is no time in which to execute. In domains with limited time budgets, the optimal behaviour of an agent is the one that maximises reward over the entire time budget, whether that means more or less time is dedicated to planning. However, all things being equal an agent should plan for the shortest time possible to achieve the maximum reward. This is important in continual domains where an agent may receive additional goals in the future. Thus, a reduction in planning time can help maximise the opportunity for attaining reward in the future. To be able to maximise reward, an agent should be able to make decisions on how to share its time budget between planning and executing. This could be in the form pre-computing an amount of time to spend planning, or deciding during the planning process when planning should stop.

Consider the motivating example of an unmanned aerial vehicle (UAV) that must make a tour of various areas and take reconnaissance photos of each area. The agent is given a fixed time budget and a set of areas to visit. The agent must both: compute a plan; and visit as many of the areas of possible – all within the fixed time budget. If planning consumes too much time then only as much of the plan that can be executed in the remaining time is executed. It is therefore important for the agent to be able to reason about the utility of planning. For planning to be worthwhile it must produce an increase in reward that is greater than any loss in reward caused by the agent being unable to fully execute its current plan. This is where the problem differs to pure planning problems in that additional planning can be *detrimental* to overall reward achieved by the agent. The agent must find the right balance between spending time producing a better plan versus executing the current plan it has and obtaining the reward for that plan.

1.1.2 What Should an Agent do When Planning?

Planning and acting are generally seen as mutually exclusive activities. To act, the agent must first plan; and if an agent is acting it is because it has already planned. Indeed, this is assumed to be the case in the previous UAV domain. Acting ideally in goal-directed

manner requires planning, or looking ahead. The further into the future the agent looks ahead and the longer the agent looks ahead for, the better a plan the agent can compute. However, if it is possible to minimise lookahead (specifically with respect to the amount of time spent planning) and still be goal-directed, then it can be possible to act whilst planning. By acting whilst planning the agent could be able to get closer to its goal, and if the agent is closer to its goal then this will reduce the amount of work required to compute plans that achieve the agent's goals. The downside of acting whilst planning is that any action undertaken cannot be provable optimal (otherwise planning would not be necessary). If the action is not optimal then it could be detrimental or prevent the possibility of executing superior plans. Having started the action, the agent will be forced to commit to plans that contain the action as the starting action. However, it could also be that starting an inferior plan earlier than a superior plan may mean that a higher overall reward is achieved.

This thesis looks at domains where a plan can be produced near instantaneously, but computing a good plan requires significant effort. In such domains, selecting a goal-directed action is trivial. Rather it is the ordering of such actions that is the focus of planning. In these domains the quality of a solution to a problem consists of the time elapsed between when the agent is presented with a goal and time at which it achieves the goal. For instance, in the previous UAV domain, any ordering of nodes is a valid plan, and selecting a goal-directed action is a simple as choosing a node that has not yet been visited. However, without planning it is not possible to know which ordering is the best, nor which action should appear first in the plan. Consider the example UAV problem in Figure 1.1. The agent starts at S and must visit all nodes. All edges take 1 time unit to traverse, except for one edge which takes 0.5 time units to traverse, but can only be traversed from S to B (it is a directed edge). The optimal plan is to avoid the directed edge, the nodes should be visited in order of $[A, B, C]$ or the reverse. In this problem B is the closest node, but not the optimal node to travel to first. However, if the time required to compute the optimal plan was greater than 0.5 time units, then travelling to

B is a better choice than doing nothing whilst planning. That is, the agent selects B to move to (as it is closest), it then plans as if it were at B, and whilst planning the agent moves to B (so that the plan will be valid). This results in the agent taking less time to complete its goal than if it had stayed where it was during planning and planned as if at S.

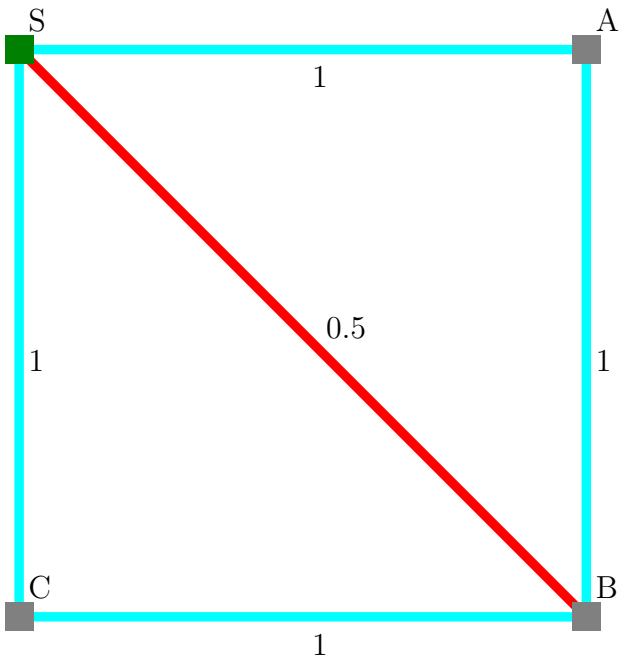

Figure 1.1: Simple UAV problem with fives edges of which one is directed ($S \rightarrow B$)

The domain also consists of multiple actors and unknown knowledge. Here, *actor* is used to mean an entity that can capable of executing actions, but has no agency of its own. A single agent controls many actors and coordinates the activities of the actors. This gives the domain concurrency, but keeps the domain single agent. The addition of unknown knowledge means that the agent must make assumptions about the state in which it is planning. If these assumptions prove to be wrong then it may be that the agent should replan. In the worst case scenario this would mean the agent needs to replan as its plan has been invalidated. In the best case scenario, it may mean further planning could result in the agent achieving it goal earlier. For instance, in a multi-UAV domain it could be found that a more or less work is required at a node. It may mean that one UAV can no longer meet its schedule if there is more work, or if there is less work then there

may exist a better way of distributing the workload amongst the UAVs. In either case replanning is required if the agent wishes to make use of the new information. However, the cost of replanning may outweigh utility gained from a better plan. Meaning that acting whilst planning could offset the cost of planning.

1.1.3 Generating Plans Quickly When Timing is an Important Factor in Behaviour

When a there is a time budget shared between planning and plan execution, if it takes too long to generate a plan and then there may not be sufficient time to execute the plan any more. In such an instance, though the plan may have been valid in at the initial time, it is not valid at the current time. Finding plans that respect internal and external deadlines *and* for which there remains enough time to execute them is more important than finding the optimal plan for the initial time and state.

A key problem with multi-agent domains is that they have a higher branching factor than single-agent domains. The planner must consider what action to execute for each agent. Thus branching factor increases from $O(a)$ to $O(a \times n)$, where a is the number of actions available to an agent, and n is the number of agents in the problem. As such, in multi-agent domains it is often more important to generate plans quickly than to generate high quality plans.

Splitting up such a multi-agent problem into a set of independent single-agent problems can allow the computation of problems to be parallelised, thus considerably reducing computation time [4, 49]. This, however, creates the new problem of how to break down a problem into these independent single-agent problems. There are two parts to this: expressing the overall goal as a set of smaller goals; and how to distribute these smaller subgoals amongst agents. The first problem is not examined by this thesis, and the second has already extensively researched (auction-theory [55, 27] and market-based goal allocation [42, 7]).

This thesis examines more complex multi-agent domains where the problem cannot

be easily decomposed into independent single-agent problems. We call these domains Heterogeneous, Cooperative and Synchronised Multi-Agent (HCSMA) domains. These domains consist of heterogeneous agents, each capable of performing a set of actions that other types of agents in the domain cannot. The goal cannot be decomposed into sets of subgoals, where each set can be achieved independently of the other sets and achieved by one type of agent. That is, no agent has all the requisite actions necessary to achieve the goal. Moreover, any valid plan will contain at least one action from one type of agent that is dependent on the action of a second (different) type of agent. Synchronisation is a requirement in domains where either one agent can undo the state created by a second agent, or where deadlines exist. In domains where agents can undo the state created by other agents then it is possible for an agent to undo a precondition to an action of another agent. Agents must synchronise such that they can complete their plans without interfering with the ability of other agents to complete their plans. In domains where deadlines exist agents must also synchronise their actions such that one agent is not delayed in completing its action because an action that is required of another agent is not completed on time.

An example of the HCSMA domain is Robocup Rescue [39]. This thesis looks at a reduced version of the problem. In Robocup Rescue, a city has been struck by an earthquake and civilians need rescuing from buildings by a deadline (separate for each civilian), and roads need to be cleared of debris. There exist two types of agent: medic and police. Medic agents are capable of rescuing civilians from buildings, and police agents are capable of clearing roads of debris. The cooperative aspect of the problem comes from the fact that agents cannot move past roads that are blocked by debris. If a medic agent is tasked with rescuing a civilian for which there exists no clear route to, then it requires the cooperation of a police agent. Each civilian has a time by which is must be rescued by, and so police must make sure there exist clear routes to civilians so that they can be rescued on time. There is no exact route that should be cleared. For instance, a longer route may be cleared as it requires less work rather than a shorter route that

requires more work. The clearing of routes is what necessitates synchronisation. There is no reward gained if a police agent clears a route to a dead civilian, or with not enough time to rescue the civilian. There is also no reward gained if a police agent clears a route to a civilian, but that no medic agent is available to rescue the civilian. Agents must cooperate and decide which civilians they will attempt to rescue.

1.2 Contributions

This section details the contributions this thesis makes to the problems areas set out in the previous section.

1. A comparison of two different ways of deciding how long to plan for. Using:

   (a) An existing algorithm (Hansen and Zilberstein [30]) to compute a fixed amount of time to plan for before planning starts.

   (b) A novel online algorithm that monitors planning and decides when to stop when it detects that the utility of planning is outweighed by the cost of time.

2. An execution framework that increases reward by acting whilst planning in domains where planning and acting have a shared time budget. It includes a collection of novel approaches to solving subproblems within this framework.

3. An algorithm that reduces the time taken to achieve a goal by enabling acting whilst planning.

4. An algorithm that manages cooperation and synchronisation in domains with multiple heterogeneous and cooperative agents.

1.3 Thesis Structure

The remaining chapters of this thesis are outlined as follows:

Chapter 2 (Related Work) discusses research literature associated with this thesis. It goes into more detail about areas of planning mentioned in the Introduction Chapter.

Chapter 3 (Optimising Planning Durations in Continual Domains) looks at how long to run an anytime planner for in a continual planning domain. It contains contributions about monitoring planning in real-time and deciding when to stop.

Chapter 4 (Acting Whilst Planning During Continuous Execution) contributes a way in which to increase reward by acting in a goal-directed manner whilst planning.

Chapter 5 (Cooperation and Synchronisation in Multi-Agent Environments) details a way in which to enable cooperation between independent agents that plan separately, but share a common goal. It also includes a way to synchronise actions within those plans in temporal domains.

Chapter 6 (Comparison of Execution Frameworks) compares the execution frameworks (EFs) discussed in Chapter 4 and Chapter 5.

Chapter 7 (Discussion) discusses the strengths and weaknesses of the approaches of described in Chapters 3 to 6, and also the merits of each approach in comparison to the others.

Chapter 8 (Future Work) describes how some of the work presented in Chapters 3 to 6 could be extended and in some instances combined. Possible issues that could arise are also discussed.

Chapter 9 (Conclusion) reiterates the contributions of the previous chapters.

1.4 Resulting Publications

The work of Chapters 3 and 4 have led to the following publications, respectively:

- Jeremy W. Baxter, Jack Hargreaves, Nick Hawes, and Rustam Stolkin. Controlling anytime scheduling of observation tasks. In *Proceedings of AI-2012, The Thirty-Second SGAI International Conference on Artificial Intelligence*, December 2012

- Jack Hargreaves and Nick Hawes. What to do whilst replanning? In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2014), UK*, 2014

CHAPTER 2

RELATED WORK

2.1 Introduction

Search is the cornerstone of planning. It is rational forethought that allows an agent to reason about the ways in which the agent is able to affect the world before carrying out any physical action. Classical planning [22] was the first form of planning, where knowledge (state) is represented as a conjunction of predicates, and an agent's possible actions consist of preconditions which must hold true in the current state and whose effects will be true of the state after the action finishes. Classical planning makes a number of assumptions to simplify planning [52]. These are:

1. That the agent knows all pertinent facts about its environment. That is there is no relevant knowledge that is hidden or uncertain.

2. The effect of actions are known and definite. This means the agent does not have to reason about action failure or having multiple possible outcomes depending on circumstances outside the agent's control.

3. All goals are to be achieved. There is no notion of partial success in completing only a subset of goals, or completing a goal only partially.

4. The world is static, and the only source of change in the world is the agent itself.

5. Goals remain constant throughout planning and execution.

6. Actions are instantaneous. Time does not exist as an explicit resource to reasoned about within classical planning. This means that actions do not have duration, or a point in time at which they are scheduled to begin or end (other than occurring directly after the last action).

These assumptions were necessary at the inception of planning research so as to make planning tractable. The planning community has since found ways to overcome these assumptions. However, many of these assumptions do not hold true in real world domains. These real world problems can consist of:

1. Temporal planning, whereby actions have durations and may occur concurrently. Typically the goal must be achieved by a deadline.

2. Incomplete or uncertain state, which means it may be unknown if the preconditions of an action are true, or what the postconditions of the action might be.

3. Dynamic worlds, where the agent must account for changes which it does not cause.

4. Multi-agent systems. Which cause a number of problems that need to be considered.

   (a) Goal decomposition – which part of the problem each agent should attempt to solve.

   (b) Coordination – agents must not invalidate the plans of other agents through their own actions. Agents should not commit to plans which prevent another agent from achieving its goals.

   (c) Cooperation – an agent should assist a second agent if the second agent is unable to achieve its goals with assistance.

   (d) Synchronisation – in temporal domains agents need to synchronise their coordination and cooperation.

5. Oversubscription – where the agent may not be able to achieve all goals. Instead the best possible subset of goals should be achieved.

The related work examined in this chapter focuses on: temporal planning problems where actions have durations and plans are measured by the time it takes to execute them; oversubscription problems where planners are not only tasked with finding a valid plan, but finding the best possible subset of goals that can be achieved; multi-agent systems where goals or tasks must distributed amongst agents; algorithms that are capable of dealing with incomplete or uncertain information; anytime algorithms that are able to compute plans almost immediately and provide a stream of plans of increasing quality; and the planning domain description language (PDDL), which standardises problem formalisations.

2.2 Temporal Planning

Temporal planning is integral to all three problems (Sections 1.1.1 to 1.1.3) looked at in this thesis. The ability of planning to reason about time is what allows agents to reason about what proportion of their time budget to allocate to planning and what proportion to allocate to execution. Without the ability to reason about how long it will take to execute a plan, the agent cannot make informed decisions about the relative utility of planning in comparison to execution. In relation to acting whilst planning the agent needs to be able reason about how it expects the state to change during planning. To do so, the agents needs to be able to reason about the duration of planning, and the duration of any actions it might execute. Without being able to reason about time the agent cannot know what will happen during planning. Without being able reason about the duration of actions this is not possible. Finally, in cooperative and synchronisation multi-agent problems, goals have deadlines. In domains with deadlines an agent must able to reason about whether a plan achieves a goal before a deadline.

Temporal planning is well researched and covers many areas [19, 20, 3, 16, 32, 9]. Of

particular relevance to this thesis are coordination and cooperation [10]. This section aims to look at the basics of temporal planning. Later sections will cover areas that incorporate temporal planning but are primarily focused on solving a different problem.

Classical planning models actions as being instantaneous – they take no time to accomplish and the effects of the action are immediate. A consequence of this approach is that the quality of a plan can only be measured based on the number of actions in the plan (the fewer actions the better). This precludes the ability to model domains that involve some temporal aspect.

By introducing the concept that actions have a duration, the metric for the optimal plan switches from the plan that achieves the goal with the fewest actions, to the plan that achieves the goal in the shortest space of time (the time from the start of the first action to the time at the end of the last action). Both types of metric can be referred to as a *makespan*, but in this section and in this thesis *makespan* refers exclusively to the temporal duration of a plan.

For the purposes of A* planning (see Section 2.7), the cost function $g(x)$, is the makespan of the actions already in the plan. The heuristic function $h(x)$ is the estimate of how much time might be needed by the additional actions that will added to the plan to achieve the goal. One way of computing this is to create a relaxed plan [1] to estimate the time remaining. The relaxed plan generates an admissible heuristic. With the addition of temporal durations it becomes possible not only to specify a goal and then optimise over makespan, but also to specify a deadline by which the plan must finish by for it to be valid.

2.2.1 Concurrency

Concurrency is a concept that is distinct from partially ordered plans [5]. A simple partially ordered plan is one in which actions have been grouped into sets where all

[1] The fast forward heuristic [36] is one way of computing such relaxed plans. The fast forward heuristic is where actions only add predicates and not delete them.

actions in one set must occur before the actions in the next set, but that there is no constraint on the ordering of actions within a set. Concurrency differs in that actions can occur simultaneously, but also that their starts and ends do not have to strictly intersect. Concepts such as 'contain', 'overlap' and 'abut' are introduced. Given two actions a_0 and a_1, with start time t_{start} and end time t_{end} then:

- Action a_0 is said to **contain** a_1 if $a_0[t_{start}] \leqslant a_1[t_{start}] \wedge a_1[t_{end}] \leqslant a_0[t_{end}]$
- Action a_0 is said to **overlap** with a_1 if $a_0[t_{start}] < a_1[t_{start}] \wedge a_0[t_{end}] < a_1[t_{end}]$ or vice versa
- Action a_0 is said to **abut** a_1 if $a_0[t_{end}] = a_1[t_{start}]$ or vice versa

These concepts allow for more precise scheduling of actions within a plan. For instance, a plane can board new passengers and refuel at the same time, but can only proceed to take-off once both boarding and refuelling have completed. In certain time-sensitive domains it may even be relevant to wait and do nothing. For example, a plane might complete boarding and refuelling early, but not immediately proceed to take-off as the runway might be in use by another plane.

Time as a Resource

Time can be considered as a resource (fluent) when planning. Like other resources it is consumed by actions. A car moving from one point to another will take time to complete the action just as it will use fuel. However, time differs from other resources in two key aspects. The first is that with other (simple) resources the total consumption is equal to the summation of the usage of the resource by each actions. However, when actions are concurrent (at least partially) then the total required time is less than the summation of action durations. This non-linearity in the consumption of time means time requires special consideration as resource. That is, the cost of executing an action is dependent on what other actions will also be executing concurrently. Conversely, temporal constraints

may mean that no action is currently possible and that the agent must wait (see the example of a plane delaying taking off in Section 2.2.1).

2.3 Over-subscription Planning

Over-subscription planning is a form of planning where it may not be possible to achieve all the goals that have been set [60]. This may be due to limited resources, conflicting requirements or limited planning time. Over-subscription planning is needed in temporal problems which have fixed time budgets, and which do not allow for all goals to be achieved. Agents must be able to reason about which subset of goals should be achieved if it cannot achieve all goals assigned to it. Limited resources are when a finite resource is required to achieve a goal (such as time or fuel), but there is not enough of the resource to achieve all goals. Goals may also have conflicting requirements, such that achieving a second goal would require the first goal to be undone. Finally, limited planning time (separate to time as resource during planning), may mean that a solution that solves all goals cannot be computed within the allotted time.

Goals can be divided into a set of strong constraints and soft constraints. Strong constraints are goals that must be achieved for a plan to be valid. Soft constraints are goals that are desirable, but not required. If a set of goals only contains soft constraints then the null (or empty) plan is a valid plan. As such, over-subscription planning fits the anytime planning domain well. That is, planning starts with the empty plan and then works to add more goals to the plan until either all goals have been achieved or planning is terminated.

The addition of soft constraints (or preferences) adds a new way in which the quality of a plan can be analysed. The quality of a plan can be measured not just by how few actions or how quickly its goals are achieved, but also the number of goals that are achieved. So whilst the null plan might be valid, it will be an extremely low quality plan. Further, each soft constraint can be assigned a cost for not achieving it (essentially a reward for

achieving the goal). This allows for stating that some goals are more important than others. Planners can use the violation cost to help direct search. A plan that achieves one important goal, could be better than a plan that achieves several less important goals.

2.3.1 Goal Selection

A new problem that is introduced by over-subscription planning is determining which set of goals an agent should plan for, and in what order the agent should attempt to achieve them. Some goals are more important than others, but may be more costly in terms of resources. This is known as net-benefit [57] – meaning the reward gained for achieving a goal minus the cost of achieving the goal. That is, a goal with a high reward may ultimately be of a lower priority because of the cost of the actions required. However, it is difficult to know this without planning first. Thus, meaning that accurate heuristics become even more important. One common heuristic is to create a relaxed plan and use its cost as an estimate to the true cost.

Accurate cost estimates are needed because, rather than planning for the set of goals that has the highest cardinality, the planner needs to plan for the set of goals with the highest net-benefit [57, 48]. If costs cannot be accurately predicted then planning may end up pursuing high reward goals that are costly to achieve and thus exhaust time planning for suboptimal set of goals. This becomes even more important in time limited anytime planning, where planning time is a limiting factor and achieving near-optimal plans early on is highly important.

One simple approach to goal selection is a greedy approach [57]. Greedy goal selection is applied by selecting the goal with the highest reward and computing the plan for that goal, then selecting the next goal and planing for that. Within this approach, there are two ways of adding goals to the plan. The first is to recommence planning with the previous plan as a start point, and the second is to replan from scratch. The benefit of the second approach is that, while it duplicates work, it may be able to find higher quality plans that use less resources. The problem with using a greedy approach is that a

high reward goal might be mutually exclusive with a set of lower reward goals that have a higher total reward.

An alternative approach [57] estimates costs of achieving goals by creating relaxed plans, and then uses these costs to create an orienteering problem to solve. An orienteering problem is where an agent must gather as much reward as possible by visiting nodes in a graph, but with a finite amount of resources (fuel/time). A planner is then tasked with achieving the goals in the solution to the orienteering problem in the order they are visited in the solution to the orienteering problem. Planning continues until all goals have been achieved or resources have been exhausted. The problem with this solution is that it is domain dependent. Goals are located in a spatial graph and do not have mutually exclusive requirements beyond the consumption of a finite resource. This has the consequence that in domains where problems contain mutually exclusive goals this algorithm may produce suboptimal subsets or orderings of goals (where an early goal prevents later goals, and that the early goal has lower net-benefit than the later goals).

Since actions cost time to execute, it becomes necessary to be able to compare time against reward. However, this issue can be side stepped by comparing plans as a tuple of duration and reward. That is, first either duration or reward is compared and if there is a tie then the other metric is considered. One difficulty is that action costs or goal rewards need to be equal or multiples of a base factor. Otherwise, the possibility of a tie is unlikely to be encountered.

In summary, goal selection is a hard problem and makes planning considerably harder as planners must now plan over 'goal space' as well as state space.

2.3.2 Time Dependent Costs

As previously mentioned, a goal can be described as necessary or desirable, or only achievable within a certain time frame. However, in certain domains it becomes useful to merge these two types of description. For example, rather than having an absolute cut off point, at which the goal becomes worthless, the goal can instead be described as having dimin-

ishing reward after a certain time. Alternatively, a goal might ultimately be necessary, but that it is desirable that it is completed sooner rather than later. This requires the merging of plan quality metrics. The quality of plan is now a composite function of the cost of time and reward for each goal, and value of time in comparison to reward needs to be taken into account. Recent work in over-subscription planning has focused on this area [8, 44].

Using continuous cost functions outperforms discrete cost deadlines in domains where the reward for achieving a goal diminishes with time [3]. Discrete cost deadlines being where several mutually exclusive goals exist with separate deadline/reward tuples. The fact that these goals are separately stated means a planner may end up attempting to pursue each separate goal, especially as they will have almost identical requirements. By instead stating reward as a function of time there will only be one goal, and it will be easier to ascertain the maximum possible reward available from a given state and thus lead to more effective pruning of the search space.

2.4 Task Allocation and Coordination

Task allocation is of significance to multi-agent problems. In these problems a goal is decomposed into a set of smaller subgoals or *tasks*. These tasks must then be allocated to the various agents in the problem. The problem of how to allocate these tasks is addressed by this area of research.

As with goal selection (mentioned in Section 2.3.1), a greedy algorithm is one way of allocating tasks. In order of decreasing reward, each task is assigned to the agent that is best suited to achieve the task [28]. However, this suffers from the same problems as any greedy algorithm suffers from. Namely, that initial allocations may be suboptimal. One difference that exists with task allocation is that it is possible to decentralise the allocation algorithm in a way that is not possible with goal selection. That is, make each agent responsible for calculating the cost of completing the task.

2.4.1 Market-based Task Allocation

Market based task allocation borrows from the idea of capitalist economic thought – that self-interested agents lead to the optimal distribution of resources [18]. The efficiency of the system can be improved as a whole if agents, with the aim of maximising individual reward, trade resources and tasks with each other. To use a market-based task allocation algorithm there are five requirements:

1. That the global (team) goal that can be decomposed into subgoals.
2. A global objective function that allows ordering of global plans.
3. An agent cost function.
4. A mapping between global objective and agent costs.
5. A mechanism for redistributing tasks/resources.

It is important that the global objective can be easily decomposed into subgoals. Market-based task allocation focuses on efficient allocations to agents, rather than the best way to decompose a problem. A global objective function is needed, so that the utility of a given global plan can be known. This enables comparisons between different global plans. An individual cost function is required so that agents can know the amount of effort that is required to undertake a subgoal. However, often computation of these costs is a hard problem in itself (requiring planning), and such costs cannot be fully known until execution (unknown or uncertain information), or subject to change during execution (dynamic environments). As such, heuristics are normally employed. The nature of heuristics means that tasks may not be optimally allocated if the heuristic is wrong. A mapping between the utility of subgoals and cost to an agent is required. It is used to show how well an agent has contributed towards the global objective.

The last part of the market-based task allocation system is a mechanism by which agents can trade subgoals and resources. Auctions are the predominant mechanism [18,

67, 38]. In an auction, goals are announced and agents invited to bid on the goal. After all bids are received or a predetermined time has elapsed the winning bid is announced. In planning domains bids are prices that the agent is willing to accept a goal for, and goals are awarded to the lowest bidder. In some instances, a reserve may be placed on a goal (that is, the maximum price that will be accepted). Different variations of bidding systems exist, including: sealed-bid versus open-cry [56]; first-price auctions [43]; versus Vickrey (second-price) auctions [61]; single-item versus combinatorial versus multi-item [62, 55]. Open-cry auctions and Vickrey auctions are more concerned with domains with adversarial agents (agents that will make untruthful bids in order to attempt to win the auction for a lower price). These auction systems are concerned with extracting truthful bids from agents. Multi-item auctions help increase the quality of task allocations. By allowing agents to create their own bundles of tasks and place bids on them, the auctioneer is better able to find the optimal allocation (that achieves maximum profit). However, computing the optimal combination of bids that achieves the highest profit is a version of the knapsack problem and NP-hard.

2.4.2 Heterogeneous Agents and Cooperation

Decentralised market task allocations can help when agents are heterogeneous. The market as a whole does not need to know about the specialisations of a particular agent. Rather it is up to the agent itself to determine whether it can or should bid for a task. However, whilst literature exists on heterogeneous agents [17], and tight coordination [27], there exists limited literature on interdependent tasks [7]. Specifically, literature that deals with interdependent tasks in domains with heterogeneous agents (where not all tasks can be solved by one type of agent). For example, where task A can be solved by agent a, but to complete task A it is required that a second (possibly resultant) task B to be completed by agent b. Task interdependence differs from tight coordination in that tight coordination focuses on the agents acting to achieve the same task. This thesis contributes a novel way of discovering and extracting such task dependencies. If further contributes

a new market allocation algorithm that prioritises allocation of such tasks in relation to the dependent task. This algorithm is extended to work with homogeneous agents and create dynamic teams assisting with different parts of the overall goal.

2.5 Continual Planning

Continual planning is an area in which agents operate with one or more of:

1. Incomplete information.
2. Dynamic environments.
3. Time pressure.

It has already been noted that time pressure is an issue of all three problem areas addressed by this thesis. In addition, the problem detailed in Section 1.1.2 has incomplete knowledge.

Incomplete or uncertain information mean that plans must either take into account different possible world states, or make certain assumptions about the world state and accept that the resulting plan may be incorrect and that replanning might be necessary. Dynamic environments mean the world state change may be subject to change outside the actions of the agent, possibly requiring replanning or accounting for this possibility. Further, these changes may be immediately known or start off as unknown and only become known when the robot directly observes that the change has occurred. Time pressures instead focus on the fact that time is shared resource between planning and execution, and that an agent must balance the need to compute higher quality plans versus the ability to execute plans and attain the reward.

Incremental search algorithms [40, 58, 41] attempt to reuse previous work when performing new searches (when it is found that the world state has changed or new information is encountered). This reduces the amount of planning needed when replanning.

Lifelong Planning A* (LPA*) [41] finds new routes from the same start node and same end node. It does so by finding which parts of the search tree are unaffected by the changed state and uses this to initialise a new A* search. Dynamic A* (D*) [45] is an online algorithm in that assumes new information is discovered via a robot's sensors. D* performs an initial A* search to find an optimal path (according to its knowledge), when new edge costs are discovered it back propagates these to effected nodes and uses this to find a new optimal path.

Reuse of the search tree is only useful if changes to the search graph are small. Significant changes would mean that there would not be much of the search tree that was reusable and the computing which parts are usable would cost more than the benefit of reusing what is recovered. These algorithms also assume that the goal remains the same, which is why parts of the search tree can be reused. This is not always the case. In the case of oversubscribed domains, it may be that new information leads to a different selections of goals and thus invalidating the entire previous search tree.

Limited work has been made at combining replanning and oversubscription problems. Cushing [15] showed that existing planning technology was capable of dealing with reselecting objectives at the replanning phase, comparing the reward gained by adding new goals into the plan versus the cost of removing current goals and commitments from the plan.

When the state changes it may or may not effect the current plan. If the change invalidates the current plan then the agent must act to find a new plan. However, if the plan is still valid then the agent is now faced with a choice. The agent can either take the opportunity to search for a new plan, on the possibility that the state change means a better plan exists [24]. However, the agent can also choose to continue carrying out its current plan on the assumption that a better plan does not exist. This is especially true in temporal domains where time is a dominant metric or deadlines exist. This is because it may be that the cost of computing the new plan is higher than the increase in reward of the new plan – and that the best course of action is continuing to execute a possibly

suboptimal plan.

2.5.1 Contingency Planning

There exist algorithms that are capable of computing more comprehensive plans that account for incomplete or uncertain information [34, 35]. Contingent planners produce plans that are trees of actions rather than a sequence. Each branch within the tree represents a divergence of belief states. That is, if an agent is unsure about the state of an object in the world it produces two or more separate belief states from the current state and then solves the planning problem for each belief state. For instance, if it is unknown whether a particular edge of a graph is traversable and the planning algorithm decides it may be worth traversing this edge, then it solves two planning problems – one in which it believes the edge to be traversable and the second where the edge is untraversable (and an alternate route must be found). Such planners may also produce graphs rather than trees. That is, the plan contains a cycle. A simple example is that, given that an action has a chance of failure, then to repeat the action until it succeeds.

One of the major benefits of contingent planning is that it is able to predict possible failure conditions, where the goal is no longer achievable, and avoid them. That is, there exists an action whose effects cannot be undone and which also makes the goal unachievable. Contingent planning accounts for all possible states of unknown variables and prioritises paths with the highest chance of success. This contrasts with replanning techniques, where once the actual value of the variable is discovered and the plan found to be invalid then it is too late.

2.5.2 Back to Replanning

Unfortunately, in many real world domains the world is highly dynamic and often only partially observable – this makes planning computationally hard. In the majority of cases this will rule out contingent planning techniques that account for all possibilities.

Instead, advances in deterministic planning have meant that a very good and naive solution to non-deterministic domains is to assume that the domain is actually deterministic [64]. When a node is expanded, the state is made deterministic by assuming the most likely state is the actual state, and then computing a relaxed planning problem to derive the heuristic value for each state. Once a plan is obtained it is then executed until either the plan is completed or an unexpected state is found – at this point replanning is triggered. This approach was state of the art in 2007, beating all other planners that actually took state uncertainty into account. This approach could be problematic when action resource consumption is uncertain. However, this could be circumvented by ignoring these discrepancies unless they invalidate the plan.

One problem with this approach is that it works less well when there is much uncertainty as the algorithm over relies on a state that is the most likely, but still improbable overall. One way of alleviating this problem is to compute the heuristic value from a selection of deterministic states that have been generated from the non-deterministic state and then take the average of the heuristic values [65].

Brenner and Nebel [6] describe a principled approach to continual planning that allows an agent to deliberately postpone parts of the planning process when it does not have enough information at hand and instead actively gather information pertinent to the plan. That is, the agent is reasoning about what information it has and does not have and what information it needs to compute a plan. Introduced are the concepts of assertions and abstract actions. An assertion is an abstract action that specifies that when certain preconditions are met then certain effects can be achieved – just like a normal action. However, the assertion itself does not achieve the effects, it just 'asserts' that they can be achieved. Rather the assertion means that once the agent has certain knowledge about the state it will be able to create a plan to achieve the desired effects. For instance, if an agent needs to enter a room and the state of the door to the room is unknown then the agent must ascertain the state of the door before being able to compute a valid plan for getting into the room. That is, if the door is open the agent can travel through the door and if the

door is closed it must be opened and before passing through. By postponing computation of solutions to these problems, it becomes possible that the agent may become stuck in a dead end. However, in complex domains where information is dynamic or partially observable the computation of contingent plans may be intractable.

In contrast to postponing planning, this thesis contributes an algorithm to bring forward plan execution into the planning process. This enables a system to resume execution immediately upon the clarification of incomplete information or an external environmental change. The parallelism of planning and plan execution allows for a more efficient use of time, thus easing time pressures that may be present. A number of strategies for selecting actions for execution during planning are presented. The best two being to either continue execution of the previous plan where possible, or to quickly generate a sub-optimal plan for use during replanning. Planning whilst the agent is actively changing state is enabled by passing to the planner the expected state for when the planner is due to finish.

2.6 Multi-agent Planning

The third problem that this thesis addresses is a multi-agent domain, and looks at how to coordinate and synchronise the activities of the agents. This section looks at other work on multi-agent problems.

Multi-agent planning is typically NP-hard ([1] as cited by [18]), and as such planning algorithms for multi-agent domains should focus on producing plans that are good enough [63] given the computational and time resources available. Indeed, the exponential blow-up of concurrent actions means that it is hard to apply state-of-art single agent algorithms to multi-agent problems [37]. Recent work on multi-agent problems has looked at how to separate multi-agent problems into multiple single agent problems that are more easily solvable [4, 14, 12, 13, 49]. Typically however, the decomposition of multi-agent planning problems into separate single agent planning problems is not easy itself. This is because there exists several types of interactions between agents that make the sepa-

ration of the planning problem complex. They are, goal selection, synchronisation and cooperation. Goal selection has been looked at in Section 2.4, so this section shall focus primarily on synchronisation and cooperation.

Brafman and Domshlak [4] look at a way of exploiting loosely coupled multi-agent planning problems (posed as Constraint Satisfaction Problems [CSP]). The state is separated into parts that an agent knows are not relevant to other agents, and parts that could be relevant – these are known as private fluents and public fluents, respectively. The example Brafman and Domshlak use is a logistics domain where trucks are moving goods from one location to another. In the logistics domain the position of the truck can be considered as private fluent. This is because the position of a truck does not effect the ability of a second truck to execute any of its actions. Therefore, an agent can reason about moving without having to inform other agents. In contrast, loading and unloading use a shared resource and so only a fixed number of trucks can be loading or unloading at any one location and time. Therefore, whether a truck is loading or unloading is a public fluent as it can effect the ability of trucks to execute their own actions and therefore accomplish the goals of the agent. Whilst enabling the problem to be posed as a series of single agent planning problems, the planner remains centralised. That is, each agent in turn solves its own planning problem, creating a series of public fluents in the process. Planning by subsequent agents must observe these public fluents and not violate them.

The work of Brafman and Domshlak [4] is mostly theoretical in nature and is not efficient in practice [49]. Nissim et al. ibid. improves on the work of Brafman and Domshlak, making it fully distributed and more efficient in practice. However, the efficacy of the work is not discussed beyond the domain of standard STRIPS problems. Neither temporal nor partial satisfaction domains are discussed. The problem of how to distribute goals between the agents in the problem is not discussed either.

Crosby and Rovatsos [12] look at multi-agent planning as heuristic search rather than as a CSP. They take from Brafman and Domshlak [4] the idea of separating out state into private and public fluents. However, instead of formulating the problem as a CSP they

formulate the problem as a search problem. Each agent builds a private planning graph and public planning graph of the states it can reach. The private planning graph consists of states that the agent is able to reach on its own. The public planning graph consists of states that the agent can reach if it relies on the (public) actions of other agents. This graph is computed using a no-delete effect heuristic for actions. Planning continues until all the agent's goals are reached either in each agent's own private planning graph or the public planning graph. Reliance on the public planning graph means cooperation is necessary. Therefore planning in the private planning graph continues to see if cooperation is indeed necessary. If an agent relies on an effect from an action in the public planning graph then that effect is added as a subgoal to the agent that added the action to the public planning graph. Full planning with the agent's updated goals rather than relaxed planning takes place to compute single agent plans, which are then subsequently merged into a single multi-agent plan.

With the exception of adding subgoals, this algorithm assumes that agents have already been assigned goals. That is, this algorithm does not look at computing a goal allocation for the agents. The domain that the algorithm works with makes classical planning assumptions with regards to time and satisfiability. That is, whilst the algorithm does analyse concurrency constraints, it only deals with the ordering of actions and not on any temporal constraints. With regards to satisfiability, the algorithm does not deal with partial satisfaction – goals are fully achieved or not at all.

Crosby et al. [14] extend the above work to include distribution of goals amongst agents by assigning goals to agents with the lowest estimated heuristic cost (using the same heuristic as before).

Crosby et al. [13] provide a further extension to their work that is able to deal with concurrent actions rather than just cooperation. Here a concurrent action is one that is performed simultaneously by multiple agents. The example domain is a grid world where all agents must get to the goal room. However, there exist transition rules between some rooms. These include: agents only being able to pass one at a time; only being able to

pass as a group (not individually); and the transition only being able to made once, but by any number of agents. Concurrent actions are split into single agent actions and the separate single agent planning problems solved. The single agent plans are then combined into a multi-agent plan about the concurrent actions in the single agent plants. The plan is also compressed so as trying to reduce global costs (the sum of the cost to each agent). This approach abstracts over time (assuming each action has a unit duration). As it stands it is unable to work with domains where actions have non-uniform durations, and where goals have deadlines.

Jonsson and Rovatsos [37] look at plan improvement in domains involving self interested agents. In the domain multiple agents attempting the same action can increase the cost of the action to all agents involved. Plan improvement involves motivating agents to cooperate so as to reduce overall action costs. The example domain they use is where an increased number of agents traversing an edge means congestion and increased costs. Agents that achieve higher reward goals by traversing the edge can incentivise other agents to use another route or wait by sharing reward. An iterative algorithm using an off-the-shelf planner is used to converge on a plan that balances all agents' preferences over plan space.

2.7 Anytime Algorithms

Reasoning about how to share a time budget between planning and execution is one of the problems addressed by this thesis. To be able to do this an agent needs to be able to constrain the amount of time a planning algorithm runs for. Anytime algorithms are planning algorithms that can be interrupted at any time and a valid plan be available. There is existing work for online monitoring and control of interruptible planning [30, 66]. Hansen (2001) uses statistical models to generate a performance profile of the planner on similar problems. This requires training on a large number of problems and is fragile to changes in problem complexity. This thesis adds to that body of work with a monitoring

"loss limiting" policy that attempts to move to plan execution after the estimated plan reward has peaked. This monitoring algorithm is fast and resistant to premature termination of planning as a result of early increases in reward during planning. It is quicker to train and resistant to changes in problem complexity.

Anytime planning algorithms produces streams of plans of increasing quality. First an initial plan is produced quickly and then improved upon as the anytime algorithm continues to work on the problem. This makes anytime planners well suited to domains where time to plan is a limiting factor.

One approach to creating an anytime algorithm is a modified version of A* [29]. This modified version of A* is called weighted A* (WA*) [51]. Weighted A* applies a weight factor w to the heuristic function of A*, where $w > 1$. This weight factor predisposes the search to expand nodes whose heuristic value is low in comparison to cost (as opposed nodes whose cost is low in comparison to heuristic value). The net effect is that, the greater the value of w the more greedy the algorithm becomes – giving preference to nodes that look close to a solution as opposed to nodes that have higher quality routes to get to the current node. WA* can be made into an anytime algorithm by continuing to run the algorithm after the first solution has been found. This allows the algorithm to continue exploring for solutions close to the first solution found, attempting to find better solutions. The reason that it is worth continuing the search after the first solution is found is because the first solution is no longer guaranteed to be optimal. This is because applying a weight factor to the heuristic value causes it to become inadmissible. That is, the heuristic may overestimate the remaining cost that is needed to achieve a solution from the current node. Instead the first solution is *w-admissible* [53] (Pearl [50] as cited by Likhachev [45]). That is, the first solution is no worse than w times the optimal solution.

Using an admissible heuristic to prune the search space, Anytime Weighted A* can ensure it does not expand states that are guaranteed not to lead to a better solution. That is, the weighted heuristic guides the order of which states to expand, but the unweighted heuristic is used to determine whether a state should be included in the search at all.

This is achieved by comparing the sum of the cost and unweighted heuristic cost of a state against the best solution found so far. If this is greater than the cost of the best solution found so far then it is not worth expanding.

2.7.1 Anytime Repairing A*

Choosing an appropriate value for the weight in anytime WA* is a problem in and of itself. With large weights, returned solutions may be considerably suboptimal, but with small weights the algorithm will take longer to find solutions. Anytime Repairing A* (ARA*) [46] works by changing the value of the weight during the course of anytime search. Unlike anytime WA*, ARA* stops as soon as it finds a solution. At this point a new lower weight is chosen. States that are still to be expanded (in the open set) have their heuristic values recalculated. Search restarts at the point, but with a new order over which to expand unexpanded states. This has the benefit of not requiring previous work to be recomputed.

2.7.2 Restarting Weighted A*

Rather than just selecting a single weight for the entire search, one possibility is to restart search each time a solution is found, but with a lower weight value. This is called Restarting Weighted A* (RWA*) [53]. Restarting each time is wasteful though – previous work must be duplicated [46]. However, the ability to choose ever lower weight values helps combat 'low-h bias'. Low-h bias is where an anytime WA* algorithm prefers exploring states close to the initial solution rather than elsewhere. Richter et al. [53] state that this is problematic when the heuristic makes early mistakes. That is, the algorithm tries to find improvements in the end part of the plan rather than attempting to improve parts of the plan that are near the beginning. By restarting with a lower weight, RWA* can pay more attention to states encountered earlier in the search. That is, as w tends to 1 RWA* becomes A*.

2.8 Planning Domain Definition Language

This thesis does not present its own planning algorithm, but rather manages planning and execution as a whole. As such, it uses off-the-shelf planners. PDDL provides a common way communicating state to a planner.

Planning Domain Definition Language (PDDL) is a problem specification language set up for the 1998 Artificial Intelligence Planning Systems (AIPS) planning competition [47]. PDDL is inspired by the STRIPS [22] syntax that was used to provide input to the STRIPS planner. It uses this syntax for both description of domains and problems within that domain. The domain specification details of the types of predicate that can exist within the domain, and the actions that can occur within the domain. Actions are described as having parameters, preconditions and effects. Parameters describe the atoms to which the action relates. Preconditions are a conjunction of predicates that must exist before the action. And effects are a conjunction of predicates that will be true after the action (effects can also include predicates that will no longer be true). Figure 2.1 shows a simple move action, with parameters, preconditions and effects.

Figure 2.1: Simple Action

```
(:action move
    :parameters (?x ?y)
    :precondition (and (room ?x)
                       (room ?y)
                       (at ?x)
                  )
    :effect (and (at ?y)
                 (not (at ?x))
            )
)
```

The aims for PDDL were to encourage empirical evaluation of planner performance and the development of a standard set up programs against which new planning algorithms could be tested against [47]. PDDL met those aims with considerable success. It became the community standard for the representation and exchange of planning do-

main models [23]. Future planning competitions extended PDDL to handle concepts not handled in its original specification. This includes: planning with resources and temporal planning [23], derived predicates and exogenous events [21], preferences [25, 26], and object fluents (non-numeric values) [33].

The first extension to PDDL added numeric values and time. Time allows actions to have durations. These durations can be static or dependent on the state of world. Figure 2.2 details a typical durative action that could be defined by PDDL2. It uses a *fluent* (discussed in the next paragraph) to describe how long the action will take. In addition it also specifies conditions and effects of the action. However, these conditions are no longer just *preconditions*, but conditions that must apply at various points in the action. Conditions can either be true at the start of the action, at the end of the action or true throughout the period that the action is being executed. This allows for a very high level of expressiveness. It becomes possible to specify that a condition must hold true at the beginning and end of an action, but that its state is not restricted during the course of the action. For instance, given an action "fly", where an aircraft flies from one airport to another, then at the beginning of the action (when taking off) everyone on board must be wearing their seatbelt, and at the end of the action (when landing) this must also be true. However, during the course of the action (whilst cruising) it is not necessary for seatbelts to be worn (only recommended). By actions having durations, a metric is introduced for measuring the quality of plan. Previously the quality of plan was measured in the number of actions contained in the plan. The less actions the better. However, now plans can be measured by their cumulative time from the start of the very first action to the end of the very last action. For instance, a plan with more actions that takes less time may be preferable to a plan with less actions that takes longer.

In PDDL, fluents are a numeric value or function tied to a predicate. For instance, in Figure 2.2 the duration of the action is based on the value of `(distance ?a ?b)`. Fluents are not just static values though, and can be changed by actions. This allows plans to take into account finite resources (such as fuel). This allows for considerations as to both:

Figure 2.2: Durative Action

```
(:durative-action travel
    :parameters (?a ?b)
    :duration (= ?duration (distance ?a ?b))
    :condition (and (at start (at ?a))
                    (over all (has-fuel))
               )
    :effect (and (at start (not (at ?a)))
                 (at end (at ?b))
            )
)
```

the possibility of the action; and whether the action is preferable to another action (or set of actions) that achieve the same effect. That is, an action might not be possible as the agent does not have enough fuel to complete the action; or that one route is preferable to another because it requires less resources. For instance, an agent might choose to drive around a hill rather than over it as this will consume less fuel even though it is a longer distance.

In PDDL3, PDDL is extended by adding the ability to express goal states as a conjunction of preferences rather than absolutes [25]. Rather than stating that there is only one acceptable sub-state that *must* exist within the final state, the goal is instead stated as any permutation of preferences, but with penalties associated with not achieving given preferences. The full conjunction of preferences remains the ideal solution, but (combined with other cost measures) may no longer be the optimal solution. The value of a given state is given by a metric in the domain – a formula that takes into account which preferences have been achieved and the values of any relevant fluents. This allows planning algorithms to consider whether a given goal is worthwhile, given the resources it takes to achieve. For example, if an agent is to optimise over reward and remaining fuel (according a given function), then the reward obtained by achieving a goal may not be worth the cost in fuel required to achieve the goal. Even in the event that there are no resources that are directly consumed it allows for anytime planning problems where the objective

is to achieve as many goals as possible within the given planning time. That is to say, the problem may be sufficiently complex so as to prevent a full solution being computed within the time constraints on planning, and so instead a plan that achieves as many preferences as possible within time set aside for planning. This is especially true of domains where time is a resource shared between planning and execution. In such domains the time required to compute the optimal plan might be mean there is not enough of the time budget left to execute the plan.

2.9 Comparison

This section provides a comparison of the work that is closest to the problems addressed by this thesis. Table 2.1 compares the properties of state-of-the-art planners and execution frameworks (frameworks that manage both planning and execution). The temporal aspect of the algorithm is split into two parts. The first is *durative*, this is whether the algorithm works on domains where actions have duration. The second is *concurrency*, this is about whether the algorithm can cope with multiple actions occurring at the same time. Even if an algorithm does not work with durative domains it can still work with concurrent domains. In such cases actions are all assumed to have the same duration. The multi-agent column refers to whether an algorithm works with domains that include multiple independent agents. [13] is a special case, as although it works with multi-agent domains, it plans serially for each agent rather than planning being distributed to each agent.

The individual contributions made by Chapters 3 to 5 each addresses one of the sub-questions laid out in Chapter 6. Chapter 3 examines how long to plan for in domains with fixed time budgets. It adds to the existing body of online monitoring and control of interruptible planning. It requires no more domain knowledge than the Anytime A* planner it uses. Using the reward function that is necessary to guide planning it can calculate the cost of planning. With this it can stop planning using a "loss-limiting" tolerance. This "loss-limiting" is domain specific, but can be calculated from training

on a few problem instances. This is an improvement on previous online monitoring and control execution frameworks [30] which require training on a much larger training set.

Chapter 4 examines the sub-question of what an agent should do whilst planning. Though there exists much work on how planning and plan execution can be interleaved to improve performance in dynamic environments and reducing the cost of replanning [40, 58, 41, 6, 45]. However, all of this work treats planning and plan execution as exclusive behaviours. Chapter 4 looks at how agents can parallelise planning and plan execution to achieve its goals faster. This is a novel concept that can lead to significant gains in reward where replanning is often required or where replanning is beneficial if the cost of time can be lowered. By acting whilst planning the cost of time can be significantly reduced.

In domains where timing is an important factor in behaviour, Chapter 5 examines whether it is better to generate high quality plans or generate plans quicker. It does so by finding ways to generate plans quicker by It does this in a domain where agents *must* cooperate to be able to achieve some goals. This domain property is independent of any time pressure, deadlines or time budget. That is, agents are only capable of executing a sub set of all actions in the domain. Therefore, two or more agents must work together to achieve some goals. This is a unique property amongst the surveyed multi-agent literature [17, 27, 7]. Market-based goal allocation is an effective way to decompose problems into sub-problems that can be solved separately as single-agent problems. However, the requirement of cooperation means this decomposition is hard. This thesis contributes a novel way of decomposing these goals into subgoals so that they can be solved as separately as single-agent problems. It further contributes a new algorithm for allocation of these subgoals that priorities allocation all the subgoals of a goal before moving on to allocation of another goal. This increases the chances of goal completion in over-subscribed domains.

Property	Benton [3]	Brenner [6]	Nissim [49]	Crosby [13]	Chapter 3	Chapter 4	Chapter 5
Planning Domain							
Temporal – Durative	✓	✗	✗	✗	✓	✓	✓
Temporal – Concurrency	✓	✓	✓	✓	✗	✓	✓
Multi-agent	✗	✓	✓	✓ [1]	✗	✗	✓
Oversubscription	✓	✗	✗	✗	✓	✗	✓
Continual	✗	✓	✗	✗	✗	✓	✓
Execution Framework							
Optimises Planning Duration	✗	✗	✗	✗	✓	✗	✗
Acting Whilst Planning	✗	✗	✗	✗	✗	✓	✗

Table 2.1: Comparison of Planners and Execution Frameworks

[1] The work by Crosby [13] is a special case, as although it works with multi-agent domains, it plans serially for each agent rather than planning being distributed to each agent.

CHAPTER 3

OPTIMISING PLANNING DURATIONS IN CONTINUAL DOMAINS

3.1 Introduction

This chapter looks at the problem of "How long should an agent plan in the presence of a fixed time budget?". Given an anytime planner, the time budget can be divided between planning and execution. Any time spent planning is time not spent executing whatever plan the agent may have. If a plan extends beyond the end of the time budget then any actions ending after the end of the time budget are not considered to have executed. Thus, the agent only gains reward for any goals it planned for if they were achieved before the end of the time budget. This prompts the question of what exactly the split between planning and execution should be.

This chapter describes a meta-management system (MMS) that decide how to allocate the limited time budget between planning and execution. This meta-management system examines the quality of plans computed by the anytime planner and attempts to stop planning as soon as possible after the optimal balance of plan quality versus available execution time is found. This MMS is compared against:

1. Several fixed planning times approaches.

2. An algorithm to pre-compute the planning time based on previous performance on

similar instances of the problem (as described by Hansen and Zilberstein [30]).

3. A hindsight approach. The hindsight approach is the theoretical best an algorithm could ever achieve. It uses hindsight to analyse the best moment when planning *should* have stopped.

We find that a dynamic planning time can help increase the amount of reward achieved by an agent. In comparison, having a fixed planning time that is good for a majority of problems, still means that there are problems for which the given planning time is sub-optimal. In relation to pre-computing the planning, it is found that it is required that the problem domain and planning algorithm pair have a predictable output of plans, and that the reward generated by those plans be predictable too. It was found that the domain that was used did not have this feature. The issues of a dynamic planning time are addressed. Namely, that off-the-shelf anytime planners are not aware of the decreasing maximum plan makespan as time passes.

3.1.1 Problem Formalisation

Anytime algorithms (see Section 2.7 in Chapter 2) provide a useful technique for trading off planning time against execution. In situations where fast, but well-supported, decisions are needed they can provide bounded optimality [54]. This requires a stopping condition for anytime planning which provides the plan with the highest reward given the time budget of an agent.

The domain that we use to explore the abstract problem is a prize-gathering orienteering problem. The problem consists of a graph of nodes. At each node is a task that can be performed to attain reward. Each task has a reward and duration. Tasks may have the different rewards and/or durations. The goal is to find the tour that maximises reward given a maximum makespan of the plan. As there is a maximum makespan of the plan it may be that not all tasks can be completed. Thus, the agent is oversubscribed.

Using an anytime planner means that a sequence of plans of increasing quality are

produced until the optimal is found. However, later plans will not have as much time to achieve reward. This is due to consumption of time by planning, which reduces the available time for execution. Thus, what would have been the optimal plan when planning commenced may not be the plan that obtains the optimal reward given the limited and shared time budget. Further, because of the shared time budget, if planning continues after a plan is produced then the full reward of that plan may not be able to be obtained if executed later. That is, if planning continues, but no new plan is found before planning stops then continual planning reduces the available time budget, reducing the amount of available time to execute the plan. This is subtly different to when computing a new plan. Here the problem is not being able to execute all the plan.

3.1.2 UAV Surveillance Tours

An instance of the prize-gathering orienteering problem is a UAV surveillance tour. The problem domain consists of planning surveillance tours where a UAV has to plan a tour of a number of locations. At each location the UAV needs to make an observation. The observation tasks have a predicted duration and reward value. The goal is to maximise the expected reward from a tour. In the UAV domain, the graph is fully connected and all edges are bidirectional.

Prize-gathering orienteering problems arise in many different situations such as: unmanned aerial vehicle mission planning, robotic security patrols and the gathering of scientific data by robotic agents.

We have applied an anytime algorithm based upon weighted A* search [59] which can plan observation schedules and can provide the optimal solution if given sufficient time and memory. In initial trial work in a UAV mission planning domain we applied a simple fixed run time limit for this anytime scheduler. This chapter compares the performance of different meta-management systems for deciding when to stop planning and start executing.

3.2 Research Approach and Methods

3.2.1 Evaluation Criteria

A meta-management system (MMS) should improve total reward achieved between the start of planning and the end of the time budget. It is not important whether the MMS spends more time planning to find a higher quality plan or starts executing a lower quality plan earlier. It is only important that whatever approach is decided should maximise reward achieved by the end of the time budget.

Reward is only obtained for actions that are completed before the end of the time budget. Partial execution of an action does not gain the agent any reward.

The MMSs will be tested against fixed planning time approaches. That is, the planner will always be run for a specified amount of time, regardless of the problem instance or the progress that the planner is making. The MMSs are also tested on different execution speeds. That is, computational resources for planning remain the same, but actions will take longer to execute. The execution speeds that an algorithm will be tested at are 1, $\frac{1}{3}$, $\frac{1}{5}$ and $\frac{1}{10}$.

Different execution speeds will change the relative 'time pressures' for planning and executing. As execution speed gets slower an MMS should change the amount of time it spends planning. As execution costs (duration) increase, it becomes more important that each action is well considered. However, reward remains the main objective to maximise, so planning for too long will reduce achievable reward.

3.2.2 Test Environment

All experiments were run on a 2.3Ghz Intel 2 Core Duo computer with 2Gb of RAM.

Real world orienteering problems do not typically contain uniformly distributed tasks. So in order to better represent real world problems, two types of problems were generated and each MMS tested on all of them. The first type used a uniform random distribution

to create a problem consisting 100 nodes. The second type contains clustered tasks. First, the 100 nodes were split into groups. Each group received a random number of nodes between 1 and 20, but such that the total number of nodes was 100. Each cluster was then given a centre (uniform random coordinates) about which the nodes would cluster. Nodes were distributed about the centre of the cluster using a normal distribution.

To find solutions an anytime planner using WA* [29] was used. The quality of the plan was based on the average reward obtained per second of the duration of the plan. The heuristic for selecting the next node to visit was the node with the highest reward as a proportion of distance. That is, the planner favours near and high reward nodes, followed by near and low, and far and high reward nodes. The planner is highly effective at finding high quality solutions very quickly, but is unable to find a provably optimal plan in a tractable amount of time. Figure 3.1 shows how plan reward increases with time (averaged over 100 runs). It should be noted that the y-axis is a proportion of the best reward found by the planner, and so has a maximum value of 1. It shows that initial improvements are found very fast and subsequent improvements diminish with time.

3.3 Meta-Management Systems to Decide When to Stop Planning

This section describes two MMSs which decide when to stop planning and start executing. One uses a predictive algorithm that uses previous observations of similar problem instances to compute the best time to plan for. This algorithm is the work of Hansen and Zilberstein [30]. The second MMS is online, and observes the planning process and decides when to stop based only on data from the current problem instance.

3.3.1 Predictive

This MMS uses the "meta-level control" framework put forth by Hansen and Zilberstein in [30]. The algorithm takes problem instances of similar complexity and assumes that

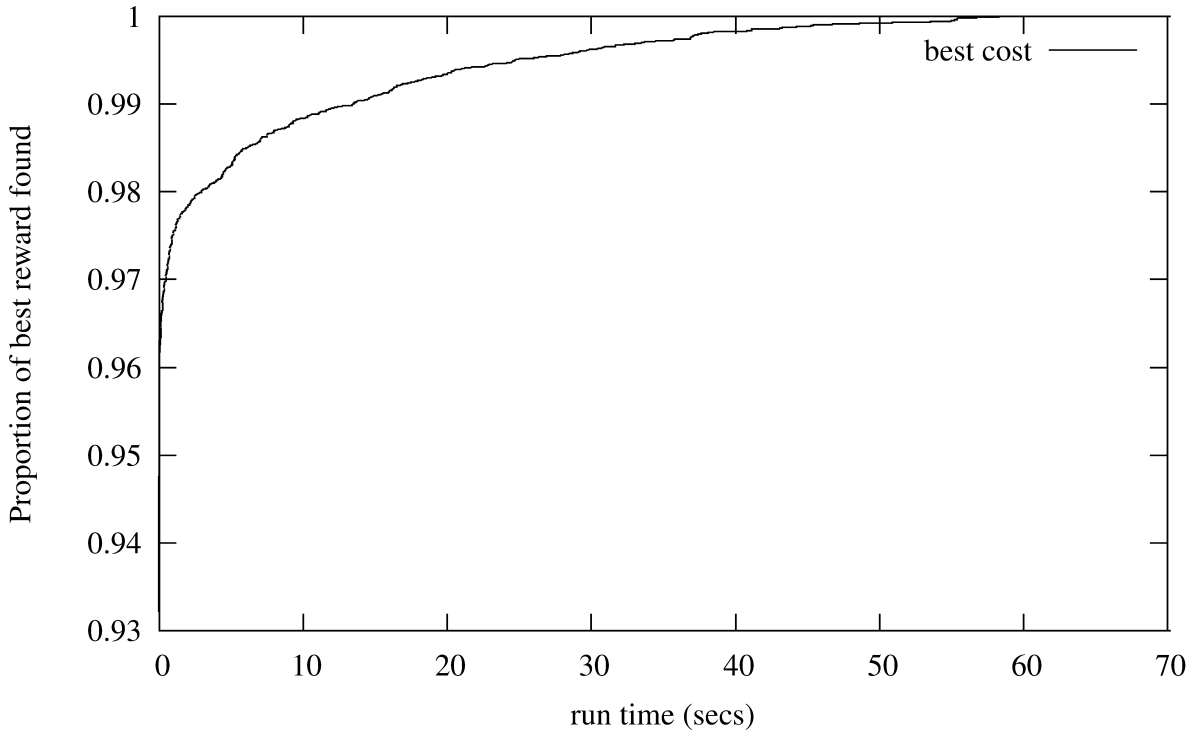

Figure 3.1: Reward proportion of best plan found against time (averaged over 100 runs)

$$t^* = \arg\max_t \sum_i Pr(q_i|t)U(q_i, t)$$

Figure 3.2: Calculating the optimal fixed allocation [30]

progress on finding better solutions can be predicted from an initial first plan. The algorithm takes the results of planning on previous problem instances and uses it to derive an estimate of the optimal amount of time to plan for, given the quality of first solution produced by planning. The formalisation of this algorithm is shown in Figure 3.2.

The optimal fixed allocation time to spend planning is represented by t^* and q is the quality of a plan. In the UAV domain quality is defined as the sum of reward gained if the plan were to be executed for the full time budget. Plan quality is represented as a discrete function and the summation is over all these discrete values. $Pr(q_i|t)$ is the probabilistic performance profile. It is the probability of getting a plan within a range of quality, given a planning time of t. $U(q_i, t)$ is a time dependent the reward (utility) function and gives the reward gained from executing a plan of quality q_i at time t – that is, if $t <> 0$, then the amount of reward gained from only the part of the plan that could

be executed within the time budget.

We modify the probabilistic performance profile $Pr(q_i|t)$ to also take the quality of the initial solution. That is, the probability of getting a plan with quality q_i given an initial plan of quality q^0 and a planning time of t – $Pr(q_i|t, q^0)$. The initial plan for the UAV is very quick to compute as it is just to visit each goal in the order they are given in the initial problem. The makespan is the sum of the time it takes to observe each goal and the time required to move between each goal (in the order specified).

In our domain we use a mix of similar, but different problem types – random distribution versus clustered distributions of tasks. This was to see how well the Hansen and Zilberstein approach was able to generalise. As such, this strategy needs an additional training set of problem instances. For the training set, 100 new problem instances were created, using the same parameters as the testing set. Planning is run for 60 seconds over each problem, with plan quality recorded at 100ms intervals.

When deciding how long to plan for on a given problem instance, the planner is first run until it finds its first plan. In the UAV domain any ordering of goals is a valid plan. Thus the initial plan is to visit the goals in the order they were specified, stopping after the full time budget is expended. As the UAV problem is oversubscribed, not all goals can be achieved. The reward achieved by this plan is then compared against the total reward available in the problem instance to estimate the progress of planning. The dynamic performance profile is used to estimate the reward that will be gained by running the planner for a range of times from 0 seconds to 60 seconds in 100ms intervals. This is compared against the cost of time which is approximated as the average reward achieved per second by the first plan. The planner is then run for the amount of time that yields the highest expected reward.

3.3.2 Loss Limiting

The loss limiting MMS attempts to execute just after execution reward has peaked. It does so online, comparing the current achievable reward (given the current plan and

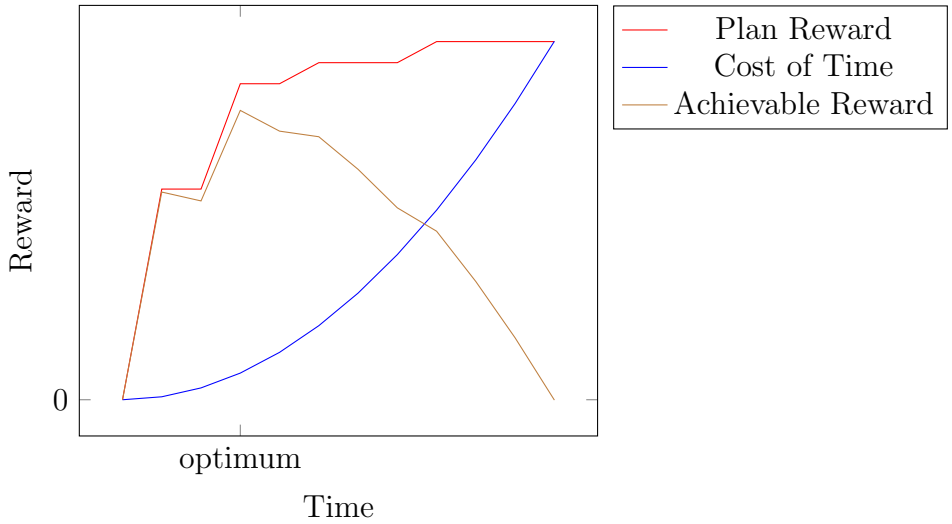

Figure 3.3: Example of net-benefit of planning in domains with shared time budgets

remaining time budget) against the maximum achievable reward seen so far. However, the MMS does not stop planning immediately as soon as achievable reward goes down so as to stop being caught in a local optima. An example of such a local optima can be seen in Figure 3.3.

The MMS decides to stop planning based on three variables. They are the reward achievable by the current plan ($reward_{now}$), the maximum achievable reward that has been seen so far ($reward_{best}$), and reward loss tolerance ($tolerance$). As time progresses it becomes possible that not all the actions in the best plan can be executed – this is because of the limited time budget that is shared between planning and execution. Planning stops once $reward_{now} < reward_{best} \times (1 - tolerance)$, and the current plan is returned. The reward achieved by this plan is $reward_{now}$. $reward_{now}$ is the sum of all reward from actions that could be completed if the plan were to be executed now.

The MMS takes three inputs: the size of the interval between when to check whether planning should be stopped ($interval$); the duration of time budget that is be shared by both planning and execution ($timebudget$); and the reward loss tolerance ($tolerance$). The MMS sets an anytime planner running, and periodically checks what the current best plan is. If the reward available for executing the current plan now ($reward_{now}$) is significantly less than any previous $reward_{now}$ then planning stops (the maximum of all

previous $reward_{now}$ is referred to as $reward_{best}$). If a new plan has been produced then it is set as the current plan. The $reward_{now}$ of the plan is calculated and compared against $reward_{best}$. Should $reward_{now}$ be better then it becomes the new $reward_{best}$. However, $reward_{now}$ is not simply the sum of reward for all actions in the plan. This is because of a diminishing time budget. Rather $reward_{now}$ is the sum of all actions in the plan that can be executed in the remaining time budget.

Algorithm 1 Loss limiting meta-management system

Require: $reward(plan, timelimit)$, a function returning the amount of reward achievable by a plan given a time limit ($reward_{now}$)
Require: $continuePlanning(time)$, a function that continues an anytime planning algorithm for the given amount of time, returning the best plan found since the beginning of planning
Require: $interval$, the interval at which it should be checked whether to stop planning or not
Require: $timebudget$, the time in which the MMS has to find a plan and execute it
Require: $tolerance$, the proportion of reward that can be lost before stopping planning

```
time_remain ← timebudget
reward_best ← 0
plan ← ∅
while time_remain > 0 do
    if reward(plan, time_remain) < reward_best × (1 − tolerance) then
        return plan
    else
        plan ← continuePlanning(interval)
        reward_best ← max(reward(plan, time_remain), reward_best)
        time_remain ← time_remain − interval
    end if
end while
return failure
        ▷ Only possible if planner does not return a plan by the end of the time budget
```

The computation required to check if planning should stop is minimal and so *interval* can be set quite low (50ms is the time used in the experiments). Whilst it is technically possible for the MMS to fail to return an executable plan, this only happens when the planner produces no plan at all by the end of the time budget.

Plan Quality Loss Tolerance

Loss tolerance is an important parameter to the MMS. It needs to be adapted to the domain and planning algorithm being used. Different domains and planning algorithms will produce different profiles of reward against time. In some domains, planning might reach near optimal plans very quickly and have lower rates of plan quality improvement compared to loss of achievable reward because of continued planning. In this case a low loss tolerance is better. If the reverse is true, in that considerable improvements in plan quality are made later in the planning process, then a higher loss tolerance is better. This will mean that the agent continues to plan even though parts of its current best plan are no longer achievable. The way in which loss tolerance was chosen for the UAV surveillance tours domain is discussed in Section 3.5.

3.4 Results

In Section 3.2.2, it was stated that varying execution speed would effect how long the agent should plan for. Figure 3.4 shows the effect of increasing time pressure on the UAV tour problem. The x axis is the runtime of the planner, and the y axis is the *scaled* reward obtained had the plan been executed at that moment in time. Reward is scaled as a proportion of the reward achieved had the best plan found after 60 seconds of planning been executed at the beginning of the time budget. This is an unbeatable baseline that no meta-management system can beat – as planning and execution to do not share a limited time budget in this baseline. The reward achieved is averaged over 100 runs. It includes four different execution speeds: 1, $\frac{1}{3}$, $\frac{1}{5}$ and $\frac{1}{10}$. The effect of decreasing execution speed means that actions now take longer to complete, but that planning is unaffected. Planning for the same amount of time still produces an identical plan of the same quality. It can be seen that at normal execution speed, planning continues to increase execution reward for 20 seconds. As execution reward increases, less of the reward of a plan is obtainable. This is because less of the actions at the end of plan can be completed. Figure 3.4 shows that

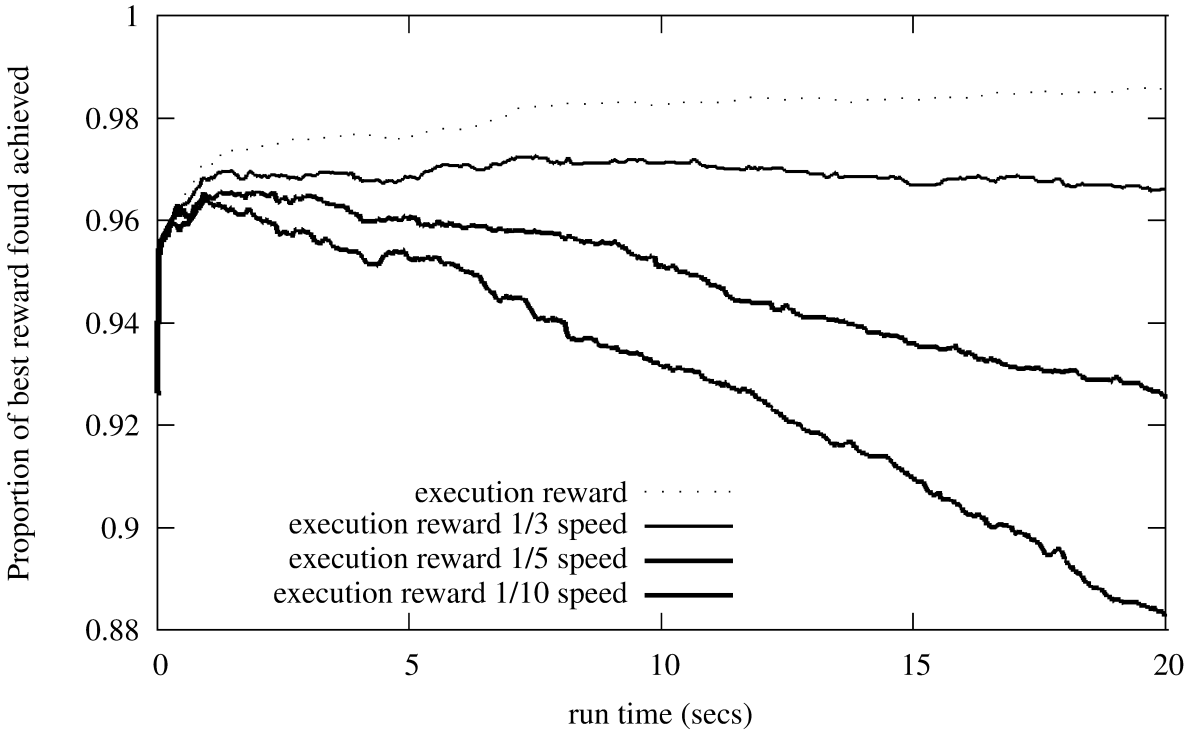

Figure 3.4: Effect of time pressure on reward achieved. Achieved reward is as a proportion of reward achieved by best plan found after 60 seconds if executed at t_0 (averaged over 100 runs).

achievable execution reward peaks earlier when execution speed is slower. As the time pressure increases, the total reward peaks much earlier and decreases swiftly, showing that the benefit of additional planning is rapidly lost. This means that planning should finish earlier as execution speed decreases.

A problem with this analysis though is that execution speed increases are simulated by taking the same plan and seeing how much reward would have been obtained if each action took proportionally longer. This does not account for that fact that planning might have produced different results if it had been aware that it had to produce a plan with a proportionally shorter makespan (that actions took longer to execute). For instance, if a planner is computing plans for an oversubscribed agent with a makespan t, but the agent only has $t-1$ to execute the plan, then not all actions can be executed. Had the planner been computing plans with makespan $t-1$ then different plans might have been produced. The heuristic used means the planner will prefer reward dense actions earlier in the plan

(reward dense meaning high reward and low duration). As such, the effects of simulating an decreased execution speed in this way are unknown. However, anytime algorithms tend to concentrate improvements to the plan later on in the plan [53]. This means that when execution speeds are increased, then improvements made to the plan are unlikely to be realised as later actions are the first to not be executed when execution speed is reduced. Finally, the planner is unaware that time is passing during in the planning process. That is, as planning progresses, the time budget decreases, and so there is no point in producing plans that have the same makespan.

Table 3.1 shows the reward obtained by each policy as proportion of the reward achieved by the best plan found after 60 seconds. For comparison, it includes three MMSs that use fixed planning times (1s, 10s and 20s). These MMSs always run the planner for the same amount of time regardless of circumstances. Hindsight is the unbeatable baseline, it shows the maximum reward obtainable had a policy picked the optimal time to plan for. That is, reward achieved by hindsight cannot be improved upon by an MMS. The only way to improve upon this result would be to improve the planning algorithm. The best policy for each execution speed is highlighted in bold. The predictive MMS does not perform well for any individual execution speed. The predictive MMS is limited because its performance is dependent on the quality of the performance profile of the problem. If a problem instance is not well represented by the average then the predictive MMS will make a bad prediction about how long to plan for. This will be especially true if the training set that the predictive MMS was trained on have a high variance (meaning very few problems are like the mean average). However, it copes relatively well with differing execution speeds. The predictive MMS loses reward more slowly than other policies as time pressure increases. This demonstrates that being able to change the amount of time allocated to planning is beneficial given different problem instances. However, the predictive MMS does not achieve the highest reward.

The loss limiting MMS performs better than the other MMSes. It is the best MMS in two of the execution reward speeds, and the second best in the other two. This is

Table 3.1: The resulting reward as a proportion of the maximum possible averaged over 100 runs for each policy with varying execution speeds

	Rate			
MMS	1	$\frac{1}{3}$	$\frac{1}{5}$	$\frac{1}{10}$
fixed 1s	0.967679	0.963957	**0.962939**	0.955168
fixed 10s	**0.970843**	0.951745	0.931818	0.881618
fixed 20s	0.965668	0.925464	0.881677	0.770558
predictive	0.966338	0.962636	0.959870	0.955899
loss limiting	0.969283	**0.965873**	0.962759	**0.957427**
hindsight	0.988795	0.978987	0.976254	0.972579

because it is better able to react to problems where significant increases in plan quality are made early on, followed by periods of minimal improvement. In this scenario the loss limiting MMS quickly finds that an optima has been reached and that planning should stop. However, the loss limiting MMS is not as good at deciding when to stop when the majority of plan quality increases are made later on in the planning process. In such a scenario, the MMS requires sustained plan quality increases to prevent it from stopping planning prematurely.

3.5 Discussion

The anytime search algorithm we used is highly efficient for finding high reward tours in the UAV surveillance problem. On average, the tours achieve 93% of total reward that was found after a minute within the first step (which typically takes 33ms). It should be noted that the agent is oversubscribed, and that there is no possible way to achieve all reward. This is further demonstrated by the fact that hindsight 'meta-management system' never achieved all the reward.

3.5.1 Predictive

The predictive MMS generally performs poorly. It is always outperformed by at least one fixed time MMS except when execution speed is 0.1 of normal speed. The predictive

MMS is limited because its performance is dependent on the quality of the improvement probability table as well as the estimates of the maximal reward and cost of time. As its future value prediction is based on averages over large numbers of problem instances it tends to perform badly in cases where the current problem instance is non typical and big improvements are made much earlier or later than the average.

However, the predictive MMS shows much better resilience to changing execution speeds than the fixed time MMSs. This is understandable given that the predictive MMS is able to respond to increasing execution time pressure by planning for a shorter amount of time. That is, as the time budget is reduced and the cost of time increases, the predictive MMS switches from planning to execution more readily if new plans are not found.

3.5.2 Loss Limiting

The loss limiting MMS outperforms the predictive MMS and is the best or second best in all cases. The loss limiting MMS is more robust to early gains (which skew the estimates of the optimal reward) but still terminates early in cases where the scheduler finds large improvements relatively late. The 99.5% loss tolerance parameter was found by trial and error, improved versions could make use of information about the variability of previous problem instances so as to not require manual training. However, the loss tolerance parameter could be reliably found from training on only a few problem instances.

One consequence of low loss tolerances for the loss limiting MMS is that planning typically stops very early. That is planning will continue until the last action in plan is no longer executable because too much time has passed. The last action in a plan will always achieve reward, because actions that increase the length of a plan without increasing the achievable reward of the plan only lower the quality of the plan. Unless loss tolerance is high, or there are many tasks achieved by the plan then usually the reward for a single task will be considerably greater than the loss tolerance. This is why planning will nearly always stop at this point.

Stopping as soon as an action drops out from a plan is a big problem with the loss limiting MMS. Anytime planning algorithms tend to explore states close to the initial solution [53]. This means improvements to the plan are more likely to be towards the end of the plan. The end of plan is also where improvements are most likely to be lost due to time pressure in the loss limiting MMS. Should the planner continue after the first action of a plan is lost then it may be that any improvements to the plan are unusable before they even found. That is, given t seconds have elapsed since planning began and that a new plan is computed that contains an action that is completed less than t seconds before the end of the time budget, then the action could never have been executed.

3.5.3 Further Work

One of the problems with the loss limiting MMS was that planning for an unknown amount of time in domains with shared time budgets creates problems with regards to the maximum makespan of a plan should be. That is, in a shared 60 second budget, if planning goes on for 10 seconds then the maximum duration a plan can have is 50 seconds. If planning continues for a total of 30 seconds then the maximum duration of the plan should be 30 seconds. Indeed, the planner has no concept that it is planning in a continual environment. This means that it cannot reason properly about a limited and decreasing time budget. The solution put forth in the loss limiting MMS is to just cut off those actions that occur after the end of the time budget. Were planning to occur for a fixed amount of time then the planner could be passed a future state rather than the current state. That is, given a planning time of 10 seconds, then the planner should be asked to produce plans with a makespan of 50 seconds rather than 60 seconds in the future.

A related problem is that it is assumed that the agent is the only source of change in the environment. Much like how the planner does not reason about how it consumes the time budget, so too does it not reason about changes to state that are not caused by the agent. Being able to predict what changes might occur and when, would be beneficial to

a continual planner.

3.6 Conclusion

This chapter examined ways of computing how long an agent should plan for (so as to optimise reward gained), given a fixed time budget shared between planning and execution. Two meta-management systems (MMS) were described that decide how much of the time budget to allocate to planning with an anytime planner.

The first is a predictive MMS that estimates the optimal planning time based on a performance profile. This performance profile is derived from the performance of the planner on previous problem instances. The second is a loss limiting MMS that monitors the progress of planning online and decides when to stop planning. The loss limiting MMS continuously monitors the amount of reward that an agent can achieve and records the maximum. Should the amount of reward an agent can achieve drop to less than a given proportion of the maximum then planning is terminated and the current plan executed.

In Section 3.5 it was noted that the predictive MMS was never as good as the loss limiting MMS. However, there are a number of limitations with the loss-limiting MMS. The main limitation relates to the fact the loss-limiting MMS is being used in a domain with a limited time budget, where further planning reduces the amount of time available to execute a plan. Since the amount of time allocated to planning is not known beforehand, then the amount of time allocated to execution is not known either. This means that the planner is unable to optimise the plans it creates for a specific length. That is, the planner may spend time optimising an end part of the plan that cannot be executed in reality as too much of the time budget has already elapsed. This has lead the work in the next chapter which instead uses a fixed planning time, but examines how reward can be increased by finding ways to execute actions during the planning process.

CHAPTER 4

ACTING WHILST PLANNING DURING CONTINUOUS EXECUTION

4.1 Introduction

Planning and execution are usually modelled as separate sub-problems. That is, planning occurs separately to execution. In the previous chapter, the problem of how to share time resources between planning and execution was examined. There the problem was how to balance the competing needs for time between planning and execution. However, this still treats planning and execution as separate processes that cannot occur simultaneously. In this chapter, the problem of what an agent should do when planning is examined. Ways in which to execute goal-directed actions during planning are explored. Acting whilst planning could be considered paradoxical in nature. Namely, planning is the act of computing a plan of action, and execution is the enacting of that plan. It is not possible to execute a plan if there is not a plan already present. However, this chapter describes multiple execution frameworks (EFs) that are capable of carrying out actions whilst planning, and that increase reward in doing so. Rather than finding an explicit way to divide time between planning and execution, these EFs seek to share the planning time with execution. In doing so, the cost of planning can be recouped (at least partially).

In domains with unknown knowledge an agent must find ways to act despite not possessing full knowledge of the state. One way of approaching this problem is to assume

default values for unknown variables and replan if the actual state is found to be otherwise. However, in problems with large amounts of unknown knowledge an agent may find itself replanning frequently. In Chapter 3 we have seen that, in domains with shared time budgets between planning and execution, both excessive planning and insufficient planning can negatively effect how much reward is achieved. It was shown that an online approach is the best way to decide how much time to allocate to planning, but that this raises its own problems in domains with limited time budgets.

This chapter presents a set of execution frameworks (EFs) that enable acting whilst planning by producing goal directed behaviours that can be executed whilst planning. These execution frameworks require the amount of time allocated to planning to be known at the start of planning – making them suitable for domains with fixed time budgets shared between planning and execution. That is, as there is a fixed planning time, the time when planning will end and execution will start is known by the planner. The planner can use this knowledge to create plans that can be executed within the remaining time budget, and that the planner can safely ignore plans that try to executed actions beyond the end of the time budget.

4.1.1 Problem Formalisation

Concurrent Temporal and Sensing Action (CTSA) problems are a sub-domain of planning problems that involve concurrent actions. The concurrency of the problem may be expressed as a single agent problem where the agent can perform multiple actions concurrently (single-agent multiple-task – SA-MT [28]). In some instances this can be a central agent that is controlling multiple actors. We use the term 'actor' to denote an isolated physical entity that can alter the world around it, but which does not have an independent decision making process (ie. it is under the control of an agent). This is in contrast to an agent which is an entity of capable of reasoning about purposeful, goal directed behaviour. Reward for a CTSA problem is measured as a function of the total time it took to achieve the goal state, from the agent being given the initial state.

In CTSA problems, the state is only partially known by the agent. That is, the agent starts with the knowledge that variables exist within the state, and the values with which the variable may take. However, the agent does not know the actual value of the variable, nor is it able to reason about the likelihood of the variable taking any particular value – this precludes probabilistic reasoning. A result of not knowing the full starting state of the problem means that either the agent must be able to make contingency plans for the various values a variable might take, or be prepared to replan during execution. For the replanning approach, the replanner must either entirely avoid the unknown variable or assume a default value. The former will lead to situations where a solvable problem is deemed unsolvable and the latter will lead to situations where the replanner produces an invalid plan or suboptimal plan. Replanning is required when new knowledge shows that a plan is invalid, but replanning could also be beneficial when the plan is shown to be suboptimal. However, this is only true if the additional reward gained from planning outweighs the cost of replanning.

CTSA problems have a limited time budget that is shared between planning and execution. This means that reward is a function of planning time, and that finding the optimal plan may not be the best course of action. That is, given a near-optimal plan returned by an anytime algorithm, the reward gained from finding the optimal plan is likely outweighed by the additional cost of the time taken to search for the optimal plan.

Finally, the agent is the only source of change and once the value of a variable is observed it will not change unless the agent directly changes the value itself.

Irreversible Actions

As stated in Section 4.1, there is a difficult to know if an action is goal-directed without first planning. Indeed, sometimes an action might be detrimental to a plan being computed by planning. In such a situation the plan would have to contain actions that reverse state changes that took place during planning. However, some actions are harder to reverse than others. Irreversible actions are actions that do not have an equal and

opposite action. An opposite action being one which is applicable in the resultant state of applying the original action, and that the effect of the applying the opposite action is to reverse the effects of the original action. An equal action is one that has identical time and resource costs. For instance, a movement action is (generally) reversible as the object that was moved can be moved back to its original position in the same amount of time. An irreversible action cannot be reversed – either at all or at least not easily. An example of an irreversible movement action would be driving off a cliff. Damage may occur to the actor that cannot be undone in the scope of the problem, and the time and effort involved in getting back to the top of the cliff is significantly higher than falling down to the bottom of the cliff.

Irreversible actions present the possibility that through an agent's own actions, the agent may permanently reduce the amount or reward available or even make the goal unachievable. This is of particular concern for many of the execution frameworks (EF) discussed herein. This is because the EFs select actions to execute that are *believed* to help achieve the goals state, as opposed to actions that are *known* to help achieve the goal state. If that belief is incorrect, then the agent may well be acting against its own interests.

4.1.2 Janitor Domain

The Janitor Domain is an example of a CTSA problem (see Appendix A for a full domain description). The problem is to ensure all rooms in a building are clean. A room has a dirtiness value that is non-negative. A dirtiness value of 0 indicates a clean room, any other value indicates a dirty room. A room may also be labelled 'extra dirty'. Extra dirty rooms require more than one actor to be present to clean the room. Cleaning extra dirty rooms is a multi-robot task as opposed to single-robot task (MR and SR respectively [28]). In the initial state given, the dirtiness value and the whether a room is 'extra dirty' is unknown. Rooms are connected to one or more other rooms. The quality of a plan is based on how quickly the goal is achieved.

In the Janitor domain there is one central agent that controls many janitor robots (actors). The actions available to agent are: to move an actor from one room to another or to clean a room. The duration of a move action is dependent on the distance between the first room and the next. The duration of a clean action is dependent on the dirtiness of a room. In the case of moving to a room with an unknown dirtiness variable then the value of that variable is immediately discovered. This will either delay the actor in case of the room being dirtier than expected or expedite the actor's schedule as less work is needed. Rooms labelled as 'extra dirty' require two actors to clean the room simultaneously. If one actor is delayed due to taking longer than expected to clean the rooms allocated to it, then this will result in delays for any actors it is coordinating with to clean extra dirty rooms.

An expansion to this problem is to make the duration of cleaning a room proportional to the number of actors cleaning the room. This means there would be a trade off between trying to discover unknown variables and acting immediately to clean rooms.

4.2 Research Approach and Methods

4.2.1 Evaluation Criteria

This section describes the metrics by which the solution (described in Section 4.3) will be evaluated. Those metrics are:

1. Completion time (total time taken to reach the goal state). This includes planning time and execution time (sometimes concurrently).

2. Number of times the planner was called.

3. Time waiting for actions to finish (act_{wait}).

4. Time waiting for planner to finish ($plan_{wait}$).

The most important metric is the makespan of the solution. That is, in the Janitor domain the most important factor is to complete the job as quickly as possible. Shorter execution times for the same problem indicate that the agent is acting in a goal-directed way during planning. Counter productive actions are ones that lead to a state that requires more actions to complete the goal that before. Shorter execution times also indicate higher quality plans are being used by the agent.

As a result of gaining new knowledge the solution may decide to replan. The number of times the planner was called indicates how often replanning occurred. Replanning can occur at most, as many times as new pieces of information are received during the problem.

Execution frameworks that *do not* allow planning and acting to occur concurrently will incur an overhead when switching from execution to planning. This is what the act_{wait} metric measures. A reduction in this number means that planning is able to start sooner after new information is discovered.

In execution frameworks that *do* allow planning and acting to occur concurrently, then it may be that there are times at which actions are not executing during planning. This potentially represents a suboptimal execution of actions whilst planning. This is what the $plan_{wait}$ metric measures. In execution frameworks that do not allow planning and acting to occur concurrently, this is simply a measure of total planning time. A value of zero would indicate that actors (as a group) were always executing at least one action whenever planning was taking place.

The EFs described in Section 4.3 are compared against a baseline that is able to pause the world during replanning. The baseline uses the same planning time as each EF. However, the baseline must still deal with unknown information. Essentially, the baseline is able to create a plan instantly when replanning is necessary.

4.2.2 Test Environment

A simulator was created to aid in carrying out plans and interrupting the plans at appropriate points when replanning was required. Since actions have durations, and actions can be executed concurrently by different actors, and events may happen during the execution of actions, the simulator must account for all of this. The simulator makes sure that the effects of an action are affected at the appropriate point during execution – actions can have effects at the beginning or end of the action. As a central controller of the world, the simulator is also responsible for making sure actions that have been scheduled by an agent are started and finished at the right point. The simulator is also decides when new information has been discovered by the agent and is responsible for dispatching this information to the agent.

Due to the heuristic nature of how some actions are chosen, it may mean that the action cannot be carried out as its preconditions are not present in the current state. As such, an EF needs to be run in a simulator that does not trust the agent to always attempt actions that are possible. Therefore, the simulator must be capable of checking the preconditions of an action against the current state, before allowing the agent to start the action.

The agent has very little control in the simulator. It is responsible for sending the schedule of the plan to the simulator, and deciding how new information effects the current plan. The agent can either continue to schedule actions in the plan or request that replanning be carried out.

4.3 Solution

In this section several SA-MT EFs are described. They manage how and when to switch between planning and execution, and schedule actions for execution during planning. In addition, several algorithms are presented that one type of EF uses to decide what actions to execute during replanning.

We introduce a way of describing a temporal state in which actions can be in the middle of execution (or partially executed). Several of the EFs exploit the ability to model a state in which an action is partially executed. This allows the EF to transition between planning and acting much more quickly and so minimise act_{wait} and $plan_{wait}$.

We also describe a way to model such partially executed actions in PDDL. This modelling is necessary otherwise the use of a specialised planner would be required that is able deal with actions that are currently mid-execution at the beginning of planning.

For planning, the EFs use an off-the-shelf planner (OPTIC [3]) that is capable of dealing with temporal problems, but not with sensing actions. The planner is called when new knowledge is obtained (this includes being given the initial state). Since the planner is unable to cope with sensing actions, all unknown variables are given a default value when the current state is given to the planner. To achieve a given goal, the EF executes a plan returned from the planner until a new observation is made. At this point the EF decides when to start replanning with respect to currently executing actions, and what actions to execute until a new plan is returned by the planner. Deciding when to replan is important as there may be sensing actions that are currently executing and whose outcomes might effect replanning. Next we list the EFs and, where applicable, the different algorithms the EFs use to decide what actions to execute whilst planning. The EFs differ in regards to: what they do when an observation is made; what they do whilst planning; and what state they pass to the planner.

Wait then Replan (WR) This EF waits for currently executing actions (at the time of sensing new knowledge) to finish and then replans.

Suspend and Replan (SR) This EF suspends actions and replans using the current state (including currently suspended actions).

Predict and Replan (PR) In PR actions are executed during the replanning process. The state that is expected after these actions have executed is given as the initial state to the planner. The following, differ in what actions are executed during

replanning.

PR-Finish Finish the currently executing actions and do not start any new actions. This is similar to WR, but planning starts earlier and with a predicted state.

PR-Reuse Schedule actions from the current plan up until the end of replanning. This includes any actions that would still be executing at the end replanning. This approach is similar to SR, but uses a predicted state rather than the current state.

PR-New Similar to the PR-Reuse, but attempts to compute a new plan quickly and use this whilst replanning instead.

Across each of the specified EFs, planning is done for a fixed amount of time. This time is set at 10 seconds for the experiments. This time was chosen as the planner was found not to produce better plans without being given a significantly larger planning time than 10 seconds.

4.3.1 Wait then Replan

Upon getting new information about the world, the agent waits for all currently executing actions to finish, whilst not starting any new actions from the plan. Once all actions have finished the planner is run with the current state for a set amount of time and then the agent begins executing the new plan that is returned by the planner. Figure 4.1 shows how, upon the completion of Actor B's sensing action, new information is discovered. However, Actor A is executing an action at that moment in time. The planner waits for this action to be completed and then passes the current state to the planner. Once planning is complete, both actors begin executing the relevant parts of their plan.

This EF adheres to the most classical planning assumptions. There is no concept of partial execution of actions. Actions are uninterruptible and their postconditions always take effect as planned for (barring external changes).

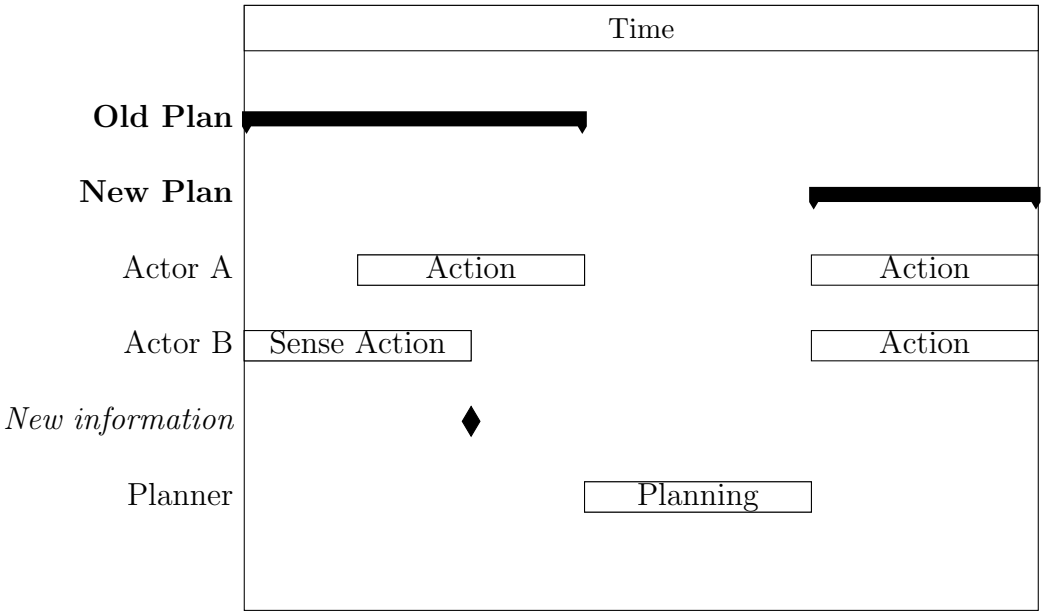

Figure 4.1: Wait then Replan

4.3.2 Suspend and Replan

This EF extends WR by eliminating the time that is spent waiting for currently executing actions to finish, so that planning can begin immediately. The algorithm assumes actions can be suspended (or interrupted). This state of suspension can be passed into the planner as an initial state to plan from. For actions whose effect is a function of time, the action is halted and the effect for the shorter duration of the action is calculated. In the Janitor domain the progress of a cleaning action is linear. If the action is halted 75% of the way through, then 75% of the dirt is removed. The remaining 25% is left to be cleaned later.

For actions that have effects that are not functions of time (i.e. actions that alter discrete values, or add/remove predicates), a temporary state is created to represent the halfway state between the beginning state of the action and the end state of the action. In the Janitor domain the effects of a suspended move action are represented this way. The position of the actor is modelled as being at a particular node. The representation of the position of a actor when its move action is suspended, is a temporary node which only it exists at, and two directed edges away from the temporary node to the start and end nodes of the original move action. This allows the janitor actor to move from the

temporary node, but not back to it. For a small example problem in the Janitor domain see Listing 4.1. The length of these edges is proportional to how far through the move the actor was when it was interrupted. For instance, in Listing 4.1 the length of the *node1-node2* edge is 10 and *temp-node* is located 2 from *node1* and 8 from *node2*. The agent runs the planner with this temporary state. Whilst planning, no actor executes any action. Execution resumes once the new plan is produced by the replanner. An example of this can be seen in Figure 4.2. As with the Figure 4.1, it is Actor B that detects new information. Rather than wait for Actor A to finish its action, the action is instead suspended by the agent and an intermediary state passed to the planner. This intermediary state is computed using Algorithm 2. In the plan that results from this replanning, the suspended action will either be completed or reversed from the temporary intermediate state.

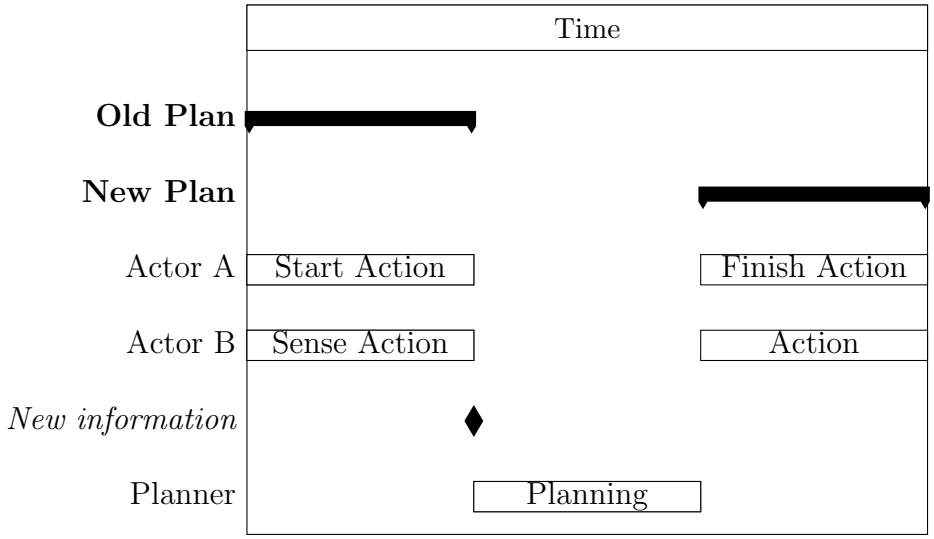

Figure 4.2: Suspend and Replan

4.3.3 Predict and Replan

The PR EF continues to execute actions whilst planning. It predicts what the state will be after planning finishes, including the effects of any actions it chooses to execute during planning, and passes along this predicted state as the initial state for planning. To be

Listing 4.1: Simple Janitor Problem with a Temporary State

```
(define (problem janitor-temp-node)
    (:domain janitor)
    (:objects actor1 node1 node2 temp-node)
    (:init
        (agent agent1)
        (node node1)
        (node node2)
        (node temp-node)

        ; undirected edge
        (edge node1 node2)
        (edge node2 node1)
        (= (distance node1 node2) 10)
        (= (distance node2 node1) 10)

        ; directed edge
        (edge temp-node node1)
        (= (distance temp-node node1) 2)

        ; directed edge
        (edge temp-node node2)
        (= (distance temp-node node2) 8)

        (available agent1)
        (at agent1 temp-node)
    )

    (:goal (and
        (at agent1 node1)
    ))
)
```

Algorithm 2 Representing a partially executed move action

Require:
$actor$, the actor that is executing the action
$move$, the action being executed
$progress$, the progress that has been made on the action (in metres)
$state$, the state at current moment in time
$location$, a unique identifier that does not already exist in the state
Assumes speed is 1 and so $distance = time$

Ensure: A state in which only two prescribed move actions are the possible actions for the given actor

$$
\begin{aligned}
& state \leftarrow state \cup \{\ at(actor, location) \\
& \wedge\ edge(location, move_{endNode}) \\
& \wedge\ edge(location, move_{startNode}) \\
& \wedge\ distance(location, move_{endNode}) = move_{duration} - progress \\
& \wedge\ distance(location, move_{startNode}) = progress\ \}
\end{aligned}
$$

able to predict the state when planning ends, PR relies on using a planner which has a predictable completion time.

A problem arises because of unknown knowledge in the state. If an observation would be made during planning the EF must model this observation in some way for the planner. The planner that is being used is not contingent, so the EF must provided an assumed value for the observation.

The problem with providing an assumed value is that this value might be wrong. In this case the predicted initial state given to the planner is wrong, and, as such, the resulting plan will be invalid. The longer the agent spends planning, the more likely it is to predict an incorrect observation. Thus an agent that uses PR must make a trade off. Does the agent: (a) spend a longer time planning, such that a better plan is produced, but at a higher risk of the plan being invalid; or (b) spend less time planning, getting a worse plan that would otherwise be produced, but at a lower risk of the plan being invalid.

PR-Finish – Predicting the End State of Currently Executing Actions

Rather than waiting for an action to finish before beginning replanning, the agent can predict what the state will be once all executing actions have finished. The agent can then use this predicted state as the initial state for replanning. This approach should beat WR as the planner can start running whilst actions are still executing, thus saving time. There is, however, the possibility that the result of an observation action might be predicted incorrectly.

This EF does not make full use of planning time to execute actions. This is because of the concurrent and temporal aspect of the problem. Different actions will have different durations, some actions will complete before others, and thus some new actions could begin before the action with the longest duration ends. This means there is further possible acting that can be done during planning. Figure 4.3 shows how planning begins before Actor A has finished acting. Note that there is still a gap between the end of current actions and the beginning of the new plan.

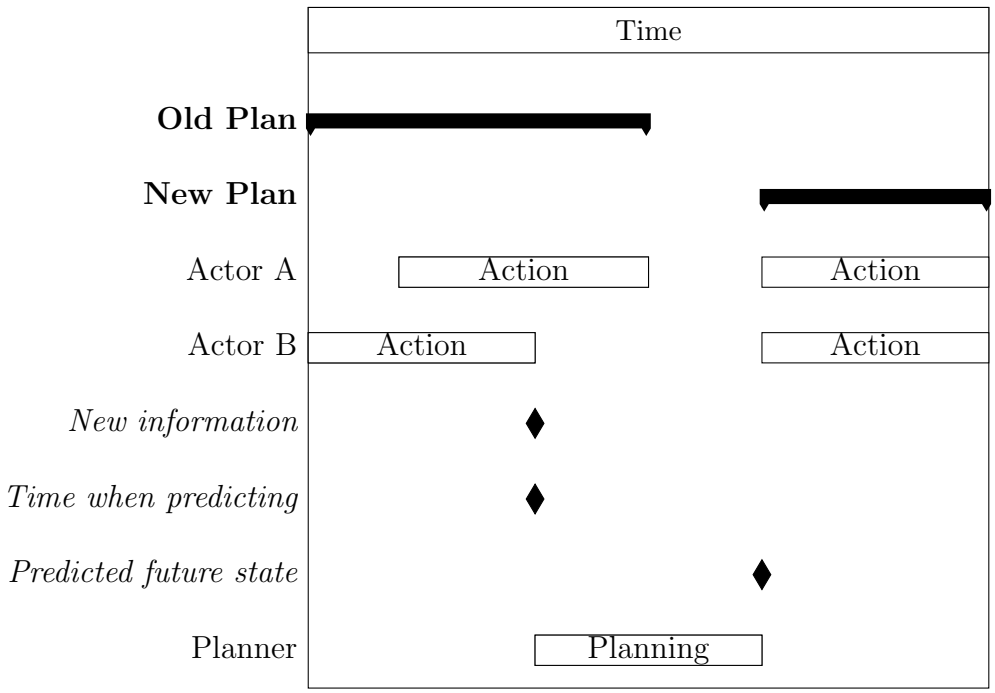

Figure 4.3: Predict and Replan (PR-Finish)

PR-Reuse – Using the Old Plan as Heuristic for Acting Whilst Replanning

This approach hinges on the idea that most new knowledge will not be significant enough to force a massive change to the overall plan. As such, the old plan remains a good heuristic as to how to act whilst replanning. The old plan is executed for as long as the amount of time given for replanning (10 seconds for the Janitor problems). Actions that cannot be fully completed before replanning completes are partially executed (as in SR). Again the predicted state for when planning would end is used as the initial state for the planner. Whilst generally a good heuristic, this approach can run into significant problems if there exist irreversible actions in the domain. That is, new information, that once reasoned about, can mean some actions in the old plan would decrease available reward or make the goal unachievable. However, by the time a new plan was obtained, that would avoid those actions, then it would be too late. In Figure 4.4 it is shown how actors continue to act using actions from the old plan whilst creation of the new plan is under way. This includes partial execution of an action by Actor A, and subsequent completion of the action in the new plan.

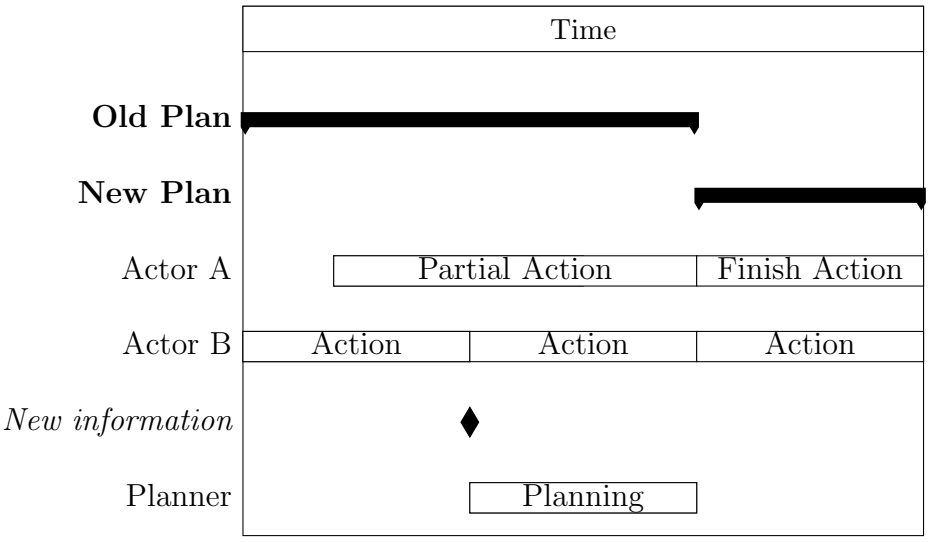

Figure 4.4: PR-Reuse

PR-New – Using a Quick Plan as a Heuristic for Acting Whilst Replanning

This algorithm takes into account that relying on the old plan is not always a good heuristic (as the old plan is out of date). The old plan might contain actions that are not possible, irrelevant, counter-productive or, even dangerous (depending on the domain). In these cases if there exists a quick way of getting a new plan, then that can be used as a heuristic. In the case of the Janitor domain, the planner (OPTIC [3]) is able to produce a plan using greedy search in a fraction of a second. Whilst these plans have makespans longer than plans produced by an anytime planner run for a longer period of time, these greedy plans are still similar in makespan to their anytime counterparts. Figure 4.5 shows how planning is split between an initial greedy plan that attempts to a find a solution as quickly as possible, and a longer planning period where the plan can be improved upon. The initial actions selected by the greedy search are likely to be in the initial actions of the new plan as anytime search tends to concentrate improvements to a plan close to the end of the plan [53]. However, actors are unable to act whilst the greedy plan is being computed.

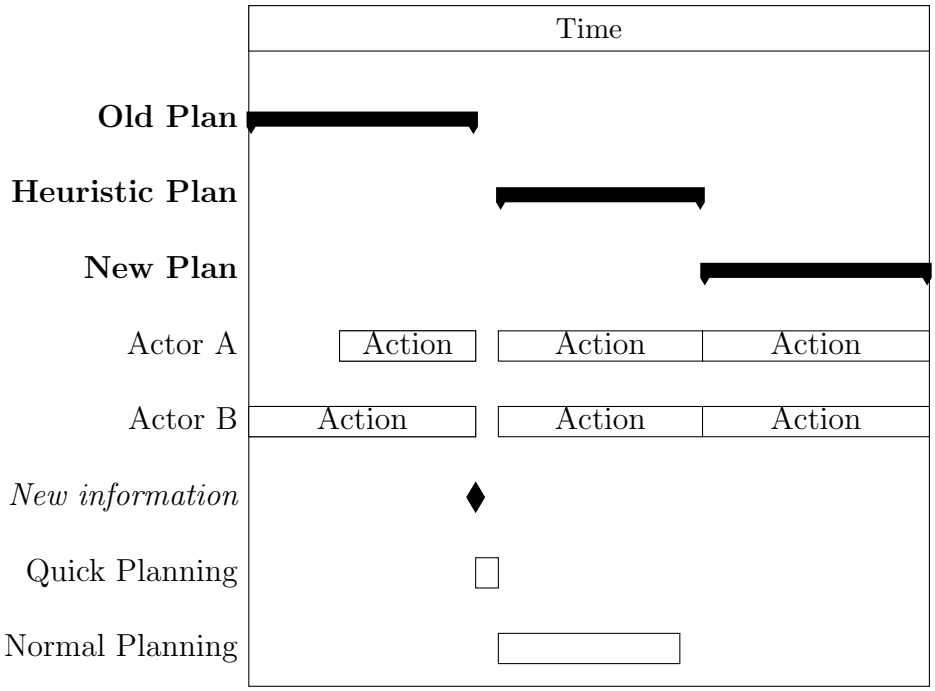

Figure 4.5: PR-New

4.3.4 Pause World and Replan

This is not a solution, but represents a baseline against which the other approaches can be compared. Essentially this approach is the same as SR except that the simulation of the world is paused whilst planning. The effect of which is that the planner is able to return a plan immediately as if it had planned for the full replanning period.

4.3.5 The Planner

All the algorithms make use of the same off-the-shelf planner (OPTIC [3]). OPTIC works on concurrent temporal problems, but has no capability to reason about unknown or uncertain states. As a result, variables with unknown values have to be given a value before the planner can be called. The specific unknown variable for the Janitor domain is dirtiness. As the agent is not aware of the distribution of dirtiness levels in the Janitor domain it was not possible for the agent to make this value the expected value. Instead an arbitrary default value is chosen. In all cases this default value was the maximum dirtiness for the problem. In practice, any non-zero value could be used. A non-zero value is needed otherwise the planner will perceive the problem as being solved (all rooms being clean). However, assuming the worst possible value means that a plan does not need to be rescheduled if the actual dirtiness of a room is less than expected. The reason for this is that the domain includes cleaning actions that require multiple actors to be present to perform the clean action. If one actor is delayed because a room it that was meant to clean is dirtier than expected, then the whole plan may be delayed as a result.

4.4 Experiments

4.4.1 Independent Variables

The Janitor domain consists of several independent variables. These are:

1. Number of rooms in problem
2. Graph configuration of rooms (room layout)
3. Number of actors
4. Range of the amount of work required to clean a room.
5. Edge length between each room
6. Number of rooms requiring cooperation between actors
7. Duration of planning

The primary independent variables are graph configuration, number of actors, and range of required work for each room. The graph configurations used are line, rectangle and square. The difference between spatial configurations is important, as it can help highlight if plans cluster actors together when cleaning rooms. For instance, if actors in a line configuration cluster, then they will have to move further to get to the next room to clean than if they had spread out. This is not as a big a problem in square and rectangle configurations as, on average, a node has a higher number of neighbours. This makes it more probable that an actor is adjacent to a node that needs to be cleaned. If shorter completion times are observed for square configurations than line configurations it means that actors are clustering.

The number of actors is useful for determining how well the EF scales. As the number of actors increases the total time to achieve the goal should decrease. This is because more actions can be done in parallel. However, increasing the number of actors also increases the branching factor during search and so if planning scales poorly, we will not see performance increases as large as in domains where planning does scale well. The number of actors in a problem is either: three, five or seven.

Each room has a uniformly randomly assigned dirtiness. This number will be within a set range. As the range increases in size, the standard deviation of assumed required

work compared to actual required work will increase. This increase means there will be more opportunity to compute superior plans that will complete the problem sooner. The distribution ranges are: 20-40, 30-60, and 40-80.

The other independent variables were kept constant. Those variables are: number of rooms in the problem, edge length between rooms, number of rooms requiring cooperation and duration of planning. The number of rooms was fixed at sixteen (16). Changing the number of the rooms in the problem would only serve to be able to look at scalability, but this is already being examined by changing the number of actors. Keeping the number of rooms fixed at 16 means that the three types of graph configuration are: 1x16, 2x8 and 4x4. Edge length was kept the same as changing it would only serve to increase average runtime. It would also serve to punish solutions that spend too much time moving to non-spatially close rooms. However, the configuration of the graph already serves this purpose. Finally, the time given for anytime planning was always ten seconds. This is because earlier experimentation showed that significantly longer planning periods (20, 30 and 60 second long planning periods were tested) did not produce better plans (with the median average reduction in plan length being 0%).

Since several of the independent variables have values assigned at random, it was decided to test the algorithms on several different variations of the problem for each configuration of the problem. That is, each configuration had 10 problems created for it. This meant there are 270 problems – ten for each of the 27 unique configurations. See Table 4.1 for each of the possible values in the 27 configurations.

4.4.2 Dependent Variables

The dependent variables against which the problems are to be analysed are:

1. Total time from start of problem to completion (runtime)

2. Total planning time

3. Goal achieved

Variable	Values
Number of Nodes Graph Layout	16 1x16, 2x8, 4x4
No. of Actors	3, 5, 7
Work Required per Room Range Work Required Distribution	20-40, 30-60, and 40-80 uniform
Edge Length Rooms Requiring Cooperation	20 3
Planning Time	10

Table 4.1: Possible values of variables, with co-varying variables grouped.

Total planning time, and runtime were recorded for each problem run for each EF. Here planning time is defined as time where the planner was running. Runtime is the time taken from starting the problem to arriving at the goal state. This includes planning time and execution time (execution time is where at least one actor is performing an action). Note that execution time plus planning time does not equal runtime. This is because some of the EFs act and plan at the same time. In addition to planning time and runtime, whether the goal was found in the final state was also recorded. This was necessary as if the planner was unable to produce a plan within the ten seconds allotted to it then the agent deemed the problem unsolvable.

4.5 Analysis

The first way of analysing the EFs was counting the number of times an EF produced the smallest runtime (and achieved the goal state). Excluding the 'Pause World and Replan' baseline, PR-New achieved the best – producing the smallest runtime 46% of the time. This was closely followed by PR-Reuse – it was best 40% of the time. This shows that acting whilst replanning is an effective way at reducing runtime. Table 4.2 shows the percentage that a given EF produced the smallest runtime for problem where all execution frameworks achieved the goal.

However, the best time saving seems to come from being able to represent partial execution states. In the case of the EFs that do act whilst planning, this allows the agent

WR	0%
SR	2%
PR-Finish	11%
PR-Reuse	40%
PR-New	47%

Table 4.2: Win percentage without 'Pause World and Replan' baseline

WR	0%
SR	0%
PR-Finish	7%
PR-Reuse	15%
PR-New	20%
Pause World and Replan	58%

Table 4.3: Win percentage with 'Pause World and Replan' baseline

to keep its actors active while waiting for an actor that is executing an action with an extended duration to finish. This is evidenced by the fact that WR *never* produced the best runtime for any problem, whilst SR produced a few of the best runtimes (2%). With respect to EF that do act whilst planning, presenting partial execution states allows the agent to act as much as is possible before planning ends. This is demonstrated by the fact that PR-Finish which does not use partial execution states is outclassed by the two EFs that do (PR-Reuse and PR-New). PR-Finish produces the best time for 11% of problems versus PR-Reuse and PR-New which produce the best time in 40% and 46% of problems respectively. This is because the agent is able to schedule all actors to be executing actions for entire duration of planning.

The competition as to which EF was best for each problem is only half the picture. Given EF A is slightly worse in a majority of cases, but significantly better in a minority of cases in comparison to EF B then A is approximately as good as B, with respect to average runtime. However, comparing the EFs only with respect to the number of times they produced the smallest runtime would show B being significantly better than A. As such, a second way of comparing the EFs is to normalise the runtime for each problem with respect to the best runtime for that problem. Runtimes were normalised using the following function.

$$normalise(p, e) = \frac{runtime(p,e)}{\min\limits_{\forall e_i \in E} runtime(p,e_i)} - 1$$

Where p is the individual problem, e is the execution framework, E is the set of all execution frameworks and $runtime(p, e)$ is the time at which the goal was achieved for a given problem and EF. An optimal EF would have a total normalised score of zero. If one EF failed to complete a problem then that problem was not included in the results. An EF that is the best for one problem, 5% slower than another EF that is best for a second problem and 10% slower for a third would have a total normalised score of 0.15.

WR	49.22
SR	38.60
PR-Finish	33.47
PR-Reuse	11.36
PR-New	3.97
Pause World and Replan	2.89

Table 4.4: Total Normalised Scores

It was expected that the 'Pause World and Replan' baseline would have a normalised score of 0, with no EF being able to produce a shorter runtime than it. However, this was not the case (as shown in Table 4.4). Of the 270 problems 93 were successfully completed by all algorithms. The total normalised scored was calculated from these 93 problems. The total normalised score for the 'Pause World and Replan' baseline was 2.89. By comparison, the normalised score for PR-New was 3.97. The highest score was for WR at 49.22. So we can see that PR-New is highly effective, and sometimes it was better than the 'Pause World and Replan' baseline, as shown in Table 4.3. Whilst PR-Reuse and PR-New were ranked closely in terms of percentage of problems with shortest runtime (40% to 46% respectively), PR-Reuse and PR-New got different total normalised scores. Indeed, the normalised score of PR-New is considerably closer to the baseline than PR-Reuse. This means that where PR-Reuse is better than PR-New the difference in runtime was small. However, where PR-New is better than PR-Reuse the difference was large.

One problem with this analysis is that in 91 of the 270 problems PR-New did not reach the goal state (and thus these problems were not counted). An EF is considered to have failed a problem if the planner cannot produce a plan within the time given. One

possible explanation could be that PR-New is not a good heuristic, and initially pushes the state away from the goal state before travelling toward the goal state again. As the actor moves further from the goal state, the anytime planner must explore more of the state space to find a solution – more than can be explored in the ten seconds given to the planner. However, this explanation seems unlikely since in the cases where PR-New does succeed it is the best or very close to the best solution in nearly all cases.

It is interesting to note that despite using the same underlying planner on the same problem, different EFs failed to reach the goal state for different problems. This is possibly explained by the fact that different EFs will provide different initial states when replanning. This is because different EFs explore more or less of the world before replanning, and so make different predictions about the future state of world when initialising the replanner. When initially deciding the planning time to use for planning, a failure rate of approximately 10% was observed and that this was the result of a few hard-to-solve configurations.

The non-zero total normalised score of the 'Pause World and Replan' baseline can be explained by the fact that the states of EFs quickly diverge as they choose different execution paths. As such, the baseline might end up in states that are more complicated, and so harder to plan for. A more complicated state would be having more actors that are in temporary states as this adds additional objects that must be considered by the planner. As such, the baseline might end up producing worse plans than the EFs.

One problem with acting whilst planning is the possibility that replanning may fail to produce a valid plan due to starting with an wrongly predicted initial state. Due to the relatively high numbers of actors and the low edge length in relation to planning time, new information was frequently gathered during the replanning phase – thus invalidating plan the replanner was working on. The effect of this was algorithms that acted whilst planning spent considerably more time replanning than other algorithms. Figure 4.6 shows planning times for each algorithm normalised in the same way runtimes for each algorithm were normalised. This shows that ability to reduce total runtime (combined

execution time and planning time) came at a cost of increasing the amount of time spent planning. However, because much of planning was done in parallel with acting this is not problematic.

WR	1.66
SR	53.82
PR-Finish	13.02
PR-Reuse	47.50
PR-New	51.58

Figure 4.6: Total Normalised Planning Scores

4.6 Conclusion

The sub-question this chapter is addressing is "What should an agent do when planning?". Chapter 3 assumes that time is an exclusive resource, and that the question to solve is how to split this finite and exclusive resource between planning and plan execution. However, time it not an exclusive resource and parallel activities can occur at the same time. This chapter shows that by changing the way an agent approaches planing it can share the time it spends planning with execution. This addresses the main question of this thesis not by finding a fixed division of time to spend planning and the rest to spend on execution, but by finding a way to parallelise the process of planning and execution.

In this chapter presents three execution frameworks that increase reward by acting whilst planning. They are PR-Finish, PR-Reuse and PR-New. By planning with a predicted future state rather than the current state these Execution Frameworks inform the planner of actions taking place during planning. This enables the agent to increase the amount of time spent executing actions, and so long as these actions are goal-directed they will increase reward. However, for the Execution Framework to accurately predict the future state the time of the state that is being predicted must be known. This is so that the EF will know which actions have completed and which are yet to start or are in the middle of execution. This means that the EF must plan for a fixed amount of time –

starting planning at the time of the current state and finishing at the time of the predicted future state. If the EF plans for less time, then the plan cannot be executed immediately as it might be contingent on the effects of actions that have not finished executing. If the EF continues to plan after the time of the predicted state then that is time wasted that could have otherwise been spent on execution. As such, the contribution of this chapter is incompatible with the main contribution of Chapter 3, which plans for a variable amount of time (based on online monitoring of the progress of planning).

The EFs parallelise planning and execution in two ways. The first is to provide a representation of actions that are in the middle of execution to the planner. This enables replanning to occur at any time, regardless of the execution state of the agent. This is important in multi-agent domains as an agent in the middle of execution may receive information from another agent or external source that invalidates the current plan of the agent or enables better plans that were previously impossible. The second way is by selecting actions to execute during the planning.

An agent can only plan if it can describe the current state to a planning algorithm. Planning languages generally do not have ways in which describe actions that have only being partially executed. And where a planning language does enable descriptions of partial execution, they focus on the consumption and production of real-valued resources. Missing from this, is an ability to describe a transient state between two predicates. This chapter describes how to take a problem description and expand it so that it is capable of describing such transient states. A transient state is one in which an actor can only take an action that will move it out of the transient state. By using transient states replanning can start immediately, regardless of any concurrently executing actions. The planner can then choose to how it wishes the actor to move out of the transient state (sometimes it may not have a choice) and then carry on planning as normal. This is especially useful in multi-actor domains where actors can operate independently, but are controlled centrally. This is because for planning to begin, all actors must finish the action they are currently executing. This takes time and can significantly slow down achieving the stated goal.

Acting whilst replanning in continual planning problems can mean higher rewards can be obtained than if nothing was done whilst planning. By using a heuristic to estimate which actions are goal-directed, and to execute these. As with using any heuristic, there is the possibility that in some problems the actions executed might actually increase the work required to achieve the goal. However, this chapter has shown that acting whilst planning can result in higher rewards (even if the actions being executed are not known to be optimal). In this chapter three ways in which an agent can continue to act whilst planning were proposed (the PR EFs in Section 4.3.3). These EFs show a decrease in runtime when compared to classical approach of separating acting and planning (WR) (expressed as a normalised score in Section 4.5).

PR-Finish and PR-Reuse operated on the assumption that individual changes in state would not significantly effect the plan, and the current plan that the agent was executing was a good heuristic on how to act whilst planning was occurring. The two EFs differ only in that the first only finished currently executing actions, whilst the second would start to execute new actions from the old plan during replanning process. In the Janitor domain that the EF was tested on this was found to be effective.

PR-New assumes that the final plan produced by planner will not differ significantly, at least in the early parts of the plan, from a greedy plan. The EF finds an initial greedy solution to the planning problem and uses this to as a heuristic on which actions to execute whilst running the main replanning task. By trading off the ability to execute for a small part of the replanning process, the agent gains the ability to act in a guaranteed goal-directed manner. This is because the information that triggered replanning may have invalidated the plan such that actions the agent would execute next are not possible. In certain domains, it would even mean avoiding actions that would make achieving the goal require more work. This EF was shown to be the most effective of the PR EFs that were tested.

4.7 Further Work

There are three main areas to extend the EFs proposed. They are the addition of: irreversible actions; dynamic goal adjustment and optional actor cooperation on actions. The most important of which is the addition of irreversible actions. PR-Finish and PR-Reuse both execute actions from the previous plan as a heuristic on how to act whilst replanning. It is assumed that these EFs would perform poorly in the presence of irreversible actions as the EF would commit to irreversible actions that will render the goal unachievable. This is because the old plan is unaware of the information that means such an action will cause the goal to become unachievable. If the agent were to replan before acting, it would find that goal cannot be achieved if this action is executed and so avoid it. This is not a problem for PR-New. This is because all actions are the result of a plan that is known to be valid given the current knowledge state. However, this does not stop PR-New producing a suboptimal plan that contains an irreversible action. That is, there may exist a better plan which becomes invalid if the irreversible action is executed. Irreversible actions could be further extended to not be completely irreversible, but rather have varying costs depending on the direction of the actions. For instance, driving down a hill requires less energy than driving up a hill. Here, driving down the hill is not irreversible (like driving off a cliff), but driving back up the hill will take longer and more fuel than driving down. It is expected that PR-New will not avoid plans that contain actions that are costly to reverse.

Irreversible actions already exist in the Janitor Domain. That is, a janitor actor has no way to 'unclean' a room. However, this is not problematic as the domain always requires that a room be cleaned and never 'dirtied'. To introduce an irreversible action with negative consequences, the clean action can given be negative side effect in specific situations. That is, the new goal is to clean all rooms unless the room was marked as 'do-not-disturb'. If a 'do-not-disturb' room was cleaned, then it would be marked as disturbed. The full new goal being:

$$\forall r \in R, (cleaned(r) \wedge \neg do\_not\_disturb(r)) \vee (do\_not\_disturb(r) \wedge \neg disturbed(r))$$

Where R is the set of all rooms. However, these 'do-not-disturb' rooms are not known in advance, and the agent will be required to make contingencies or replan as and when these rooms are discovered.

4.7.1 Dynamic Goal Adjustment Whilst Executing

The current CTSA problem only allows for a fixed goal, and that goals do not have deadlines by which they must be completed. An addition to the CTSA problem would be to have deadlines for individual goals, and to allow the addition of new goals as time progressed. This allows CTSA problems to represent continual and extended execution domains. An example of which is modern warehouses. Some modern warehouses dispatch goods for e-commerce websites on the same day as the goods are ordered. This requires packing and shipping EFs to be able to respond to new goals throughout the day and provide plans that will ship as many goods as possible as quickly as possible.

It would not really be possible for a contingent planner to create a policy that would be robust to the addition of new goals. This is because the planner would have to predict when a new goal would be added and what form the goal might take. This would massively increase the plan space that the planner had to search over. Quantising states and times would simplify matters and might make contingent planning possible. However, replanning might be a better option due to decreased complexity of the problem (not having to optimise over multiple branches with plans for different goals).

4.7.2 Optional Actor Cooperation on Actions

One last interesting extension to the CTSA problem is to permit optional cooperation of actors on an action. This stands in contrast to forcing cooperation as in the case of 'extra dirty' rooms. With optional cooperation the duration of an action is reduced with the number of actors working on it. Each additional actor should produce diminishing reductions in duration. This presents an interesting interaction between how to distribute

actors, and where the highest workload is in a problem. In a problem with a uniformly distributed workload, then the optimal plan would be to evenly distribute actors throughout the problem. Whereas, if there were concentrated patches of work, then the optimal plan might be to concentrate actors in these areas and send smaller groups of actors or single actors to do the work in areas with smaller workloads.

CHAPTER 5

COOPERATION AND SYNCHRONISATION IN MULTI-AGENT ENVIRONMENTS

5.1 Introduction

In domains where time is shared between planning and plan execution, being able to create plans that valid for the time when they are scheduled to be executed is important. This can be the result of deadlines or external events that change the state of world. These changes can mean that a plan created with respect to an initial state is no longer valid for the current state (after planning has finished). In domains where finding high quality plans can take a substantial amount of time, this means that finding plans quickly is more important. By finding plans more quickly more time can be dedicated to plan execution and observing constraints based on deadlines or the occurrence of foreseen external events.

Multi-agent planning is NP-hard [1]. As a result generating high quality plans can take a long time. An execution framework (EF) working on a multi-agent domain should focus on computing plans that are good enough rather than computing the optimal plan [63]. Centralised, single-agent planning algorithms are not well equipped to deal with the exponential blow-up of plan space caused by concurrency of actions in multi-agent domains (which increases branching factor) [37]. Decentralising the planning process can help mitigate this problem.

Decomposing a multi-agent problem into a set of single-agent problems is an effective

way of achieving this [13]. However, in domains that have task interdependence [7], such decompositions are not simple. Task interdependence is where the goal is for a given task to be completed, but for that task to be completed a second (possibly unspecified) task must first be completed. It may be that there is no such agent that is capable of completing both tasks in domains with heterogeneous agents. In multi-agent domains with deadlines there is the additional problem how to coordinate and synchronise the agents quickly and to make sure that all tasks, including dependent ones are completed by the deadline.

This chapter presents an execution framework for decomposing a multi-agent problem with deadlines into a set of partially interdependent single-agent problems that can be partially solved in parallel. This approach is found to be significantly better than a centralised approach. The decomposition of goals identifies key sub goals around which synchronisation must occur. By identifying these subgoals, an execution framework can present a simpler problems for the planner to solve, and in doing so, helps the planner focus computation resources on the most important parts of the overall problem.

5.1.1 Problem Formalisation

Heterogeneous, Cooperative and Synchronised Multi-Agent (HCSMA) domains are composed of specialised heterogeneous agents that are given a goal that requires more than one type of agent to take part. Each type of agent has a set of actions that only that type of agent is capable of. Goals are posed such that the unique actions of each agent type are required to achieve the goal. Generally, this means that the goal is dependent on the postconditions of the action(s) of one type of agent, and the preconditions of those actions are dependent on the postconditions of the action(s) of another type of agent. In this chapter, this is what is meant by cooperation amongst agents

Individually, agents may either be: incapable of achieving the goal state from the current state; or incapable of achieving the goal state from any state. However, together the agents are capable of achieving the goal. A basic example can be seen in Figure 5.1.

1. Initial state
2. Agent a executes action x
3. Intermediary state
4. Agent b executes action y
5. Goal state

Figure 5.1: Basic HCSMA Example

Agent b can only achieve the goal state from the intermediary state, but cannot reach the intermediary state from the initial state. It is agent a that can perform action x that achieves the intermediary state from the initial state.

In addition to cooperation, the agents are also required to synchronise actions with each other. This requirement is caused by goals having deadlines. That is, not only must agent a executes action x, but that agent a must execute action x early enough that agent b can execute action y. A further problem with cooperation (that is not examined by this thesis), is that agents can actively work against each other (either by design or by accident). That is, an agent, through the course of its actions, can alter the state in way that prevents the goals of another agent from being able to be completed. For example, agent a might require a predicate to be true from t_n to t_{n+m} in order to complete its goal. However, another agent through the course of its actions makes the predicate false at some point between t_n to t_{n+m}, or before t_n and no agent endeavours to make the predicate true again before t_n. To simplify the HCSMA domain it is assumed that agents cannot negate states that other agents require. Instead, the focus of the domain is how an agent can positively affect the ability of other agents to achieve their goals. That is, agents are concerned with: which agents to cooperate with, and any deadlines for such cooperation.

5.1.2 Robocup Rescue

Robocup Rescue (or RR) [39] is the concrete instantiation of the HCSMA domain used in this chapter. RR focuses on how best to coordinate a group of independent and heterogeneous agents in the aftermath of an earthquake. The goal is to rescue civilians and extinguish fires. However, collapsed buildings means some roads are blocked and impassable. There are three types of agent: a medic agent that is able to rescue civilians from rubble and transport civilians to hospitals, a fire agent that is able to extinguish fires, and a police agent that is able to clear roads. The RR problem has many features:

1. Unknown and uncertain state.
2. A limited communication network.
3. Traffic management.
4. Agent and civilian health.
5. Fire spread management.

However, this thesis uses a simplified problem. This is to simplify analysis of the solution to the HCSMA domain that this thesis proposes. This means analysis can concentrate on the effectiveness of cooperation and synchronisation. As such, we exclude fire simulation, traffic management, unknown state and uncertain state from the problem. As a result only medic and police agents remain in the problem; edges in the graph have infinite capacity for agents to move across; and agents are omniscient. This leaves any solution to decide on: the order in which to rescue civilians, and which edges to clear and in which order. For the planning domain that is used see Appendix B.

5.2 Research Approach and Methods

5.2.1 Evaluation Criteria

There are three major metrics by which to evaluate the solution that this thesis proposes:

1. Number of civilians rescued
2. Total planning time
   (a) Allocation time
3. Time to complete problem (including allocation time, planning time and execution time)

The most important of which is the number of civilians rescued. In RR problems, more civilians rescued will always dominate any other criteria. However, in other problems this may not be so important. For instance, ensuring that a certain number of goals are achieved may be enough.

A more domain independent metric is the amount of time spent planning. This can be used to show how well the algorithm scales given larger and larger problems. Scalability is a highly important quality to have as it shows the algorithm to be applicable to a wide range of real world problems. In multi-agent domains, decomposing the problem into single-agent planning problems is an important problem. Allocation time will show how quickly the solution is able to do this.

Time to complete the problem helps demonstrate the quality of the plans produced. If two plans achieve the same goals, but one plan has a shorter makespan than the other, then the faster plan is better because it leaves time for new goals to be achieved or time to replan in case of unexpected world state changes from the environment.

5.2.2 Test Environment

The simulator from Chapter 4 is reused, but modified to work with multi-agent domains. As such there is no longer a central agent in control, instead agents are decentralised. Agents are now responsible for deciding goal allocations, any cooperation and synchronisation between each other, and planning to achieve their goals. The simulator retains its role in ensuring actions occur then they are scheduled.

5.3 Solution

Our novel solution to HCSMA problems enables cooperation and synchronisation by allowing an agent to bid for tasks it cannot achieve by itself. The agent can then 'subcontract' parts of the task it cannot achieve itself. The agent identifies what part of the task it cannot complete itself and issues new tasks, that if completed, will mean it can achieve its task. If the agent must complete its task by a deadline it can add appropriate deadlines to its subtasks. In HCSMA problems a task T is a 3-tuple of the predicate to be achieved, the deadline by which that predicate should be achieved by, and the reward associated with achieving the predicate by the deadline.

5.3.1 Task Allocation

First the goal, posed as a conjunction of predicates with associated deadlines, is split into individual predicates, and each predicate made a task. Tasks are allocated by auction [18, 42, 7] in order of ascending deadlines until all tasks have been successfully allocated or no agent bids for the current task. If not all tasks are allocated, then allocation of any remaining tasks will start afresh once the agents have achieved the tasks that they were allocated. Once all agents have made a bid, the task is awarded to the agent with the best bid. Agents are compelled to make a bid for a task if they have the capacity to achieve the task.

To compute the value of the bid (or cost to the agent) that an agent will make for a given task the agent plans to see if it can achieve the task and any tasks that it has already been allocated. If the agent can achieve the task then it uses the makespan of the plan as the value of its bid for the task. Using the makespan of the plan assists with load balancing. Agents with many allocated tasks will have a higher makespan than an agent with less allocated tasks. If the agent cannot find a plan that achieves the task then it does not bid.

When bidding for a task the agent does not plan with the full state. Instead the agent creates a relaxed plan whereby any action preconditions it cannot achieve itself it ignores. If the relaxed plan that the agent bases its bid off contains actions that ignored any preconditions then the agent issues each precondition as a separate new task. Should the task that is being allocated have a deadline then this deadline is back-propagated to the new subtask. That is, the new subtask is given the maximum possible deadline for it still to be helpful in achieving the task that is being allocated (according to the relaxed plan).

5.3.2 Planning

Agents then start planning once task allocation is finished. However, not all agents begin planning immediately. Should an agent have issued any subtasks, it waits for agents that were allocated to finish planning before planning itself. This is due to two factors. The first is that the problem is oversubscribed, and so not all tasks may be achieved. An agent needs to see which of its subtasks are achieved before beginning to plan itself. In an RR problem a police agent never issues subtasks and so can begin planning immediately. A medic agent only ever has its subtasks allocated to police agents and so, at most, have to wait for all police agents to finish planning before starting to plan itself. The other reason for agents to wait is to see *when* their subtasks will be achieved. The way in which deadlines are assigned to subtasks does not guarantee that an agent will be able to achieve all its tasks if all its subtasks are achieved. That is, if all an agent's subtasks were

achieved right on their respective deadlines this might not leave the agent enough time to achieve all its tasks. Waiting allows the agent to see when a subtask will be achieved and so which of its tasks be achieved earlier on in the plan.

So far the reward for achieving tasks and subtasks has not been discussed. An agent has no reason to value one task over another when planning. However, by valuing each task differently we can inform the planner of which tasks have a higher utility. In the RR problem we assign unit reward to each civilian that is to be rescued. When an agent issues new subtasks it assigns each subtask a reward based on the number of subtasks that are to be issued. That is, if only one subtask is required then the agent assigns the entire reward for achieving the task to be allocated to the subtask. However, should there be more than one subtask then reward is split evenly between each subtask.

The time required for task allocation will increase linearly with the number of tasks to be allocated, but the time allocated to planning is independent of the number of tasks to allocate. As such, less time is assigned to the relaxed planning done by task allocation than is assigned to the planning done for task planning. This will help reduce the amount of time required by task allocation.

5.3.3 Algorithms

This section details how various parts of the solution, described in the previous section, are achieved. The algorithms described are: the task allocation algorithm; the task cost function; and the task planning algorithm. Here the purpose of each algorithm or function is briefly reiterated. The task allocation algorithm is how tasks are allocated amongst agents. The task cost function is responsible for calculating the values of the bids that an agent makes. And the planning algorithm shows how each agent coordinates its planning with other agents (once the task allocation algorithm is complete).

In the following algorithms, values are objects (or records). Records have various properties that can be referenced using a subscript notation. For example, $agent_{tasks}$ means the *tasks* property of the *agent* record – the tasks that an agent has been assigned

to complete. Functions that are followed by a subscript are used to indicate a specialised version of that function. For example, $plan_{relaxed}$ is a specialised form of planning that ignores the preconditions of actions.

5.3.4 Task Allocation Algorithm

Algorithm 3 Task Allocation algorithm

Require: $tasks$, A set of tasks to be achieved
Require: $agents$, A set of agents to assign tasks to
Require: $subtasks(task)$, A function that returns the subtasks of a task
  while $tasks \neq \emptyset$ **do**
    $task \leftarrow \arg\min_{t \in tasks} deadline(t)$
    $tasks \leftarrow tasks - \{task\}$
    $agent \leftarrow \arg\min_{a \in agents} cost(a, task)$
    $agent_{tasks} \leftarrow agent_{tasks} \cup \{task\}$
    $plan \leftarrow plan_{relaxed}(agent, task)$
    if $\{subtasks(task) \mid task \in plan\} \neq \emptyset$ **then**
      $tasks \leftarrow tasks \cup \{subtasks(task) \mid task \in plan\}$
    end if
  end while

Task Cost Algorithm

A relaxed plan is one that has relaxed constraints [36]. For the purposes of this algorithm, this means that an agent ignores preconditions of actions it cannot achieve itself. For instance, the medic agent no longer requires a road to be unblocked to traverse it.

Calculating subtasks Any preconditions that were ignored as part of the creation of a relaxed plan are added to the set of tasks to be achieved as a new subtask. The deadline for such a subtask is the last possible moment for which the predicates of the requirements can remain false, but that the task still be achievable.

$makespan(plan)$ is a function that returns the duration of time between now and the last action in the plan. This is the time it would take to complete the plan were it to start immediately. This is subtly different to the difference between the first start time

and the last end time, because the plan might not schedule actions to start immediately. The function is defined as such:

$$makespan(plan) := max(\{action_{endtime} \mid action \in plan\}) - t_{now}$$

Algorithm 4 Task cost algorithm

Require: $task$, the task that is being allocated
Require: $agent$, the agent for which the cost of the task is to be calculated
Ensure: $0 \leq cost \leq \infty$

```
if task ∈ agent_tasks then
    return 0
else
    plan ← plan_relaxed(agent, task)
    if plan = ∅ then
        return ∞
    else
        return makespan(plan)
    end if
end if
```

5.3.5 Task Planning Algorithm

In this algorithm, $a_{planning}$ is a boolean condition as to whether the agent is currently planning or not. $achieves(plan, task)$ is a function that returns a boolean as to whether the plan achieves the given task. This function is necessary due to the oversubscription of the planning algorithm (the plan is not required to achieve all, or indeed any, tasks).

5.4 Results

5.4.1 Independent Variables

Of the experiments conducted there were a number of independent variables. Those being:

1. Time given to agents to come up with plans for their allocated tasks (planning time)
2. Time given to agents to compute a heuristic cost when allocating tasks (heuristic time)

Algorithm 5 Task planning algorithm

Require: *agent*, the agent to which tasks are assigned
Require: *tasks*, the set of tasks to achieve
Require: *subtasks*, the set of subtasks needed to complete *tasks*
Require: *events*, a global set of external events where each event is a state preposition and a time at which that state preposition becomes true

wait until $subtasks = \emptyset \vee \nexists a \in agents : a_{planning}$
▷ an agent is planning if it is not waiting and has not finished planning
if $subtasks \neq \emptyset$ **then** ▷ Abandon any tasks with unmet subtasks
$tasks \leftarrow \{t \mid t \in tasks \wedge (t_{subtasks} \cap subtasks) = \emptyset\}$
end if
$plan \leftarrow plan(agent, tasks, state_{initial}, events)$
$events \leftarrow events \cup \{t \mid t \in tasks \wedge achieves(plan, t)\}$

3. Number of nodes in the graph (locations that the agents and civilians can exist at).
4. Layout/connectedness of nodes in the graph (how nodes were arranged spatially, and which other nodes they were connected to)
5. The time it takes to traverse an edge from one node to another (edge length)
6. Which edges were initially un-traversable (blocked)
7. The time it takes to make a blocked edge traversable (blocked-ness)
8. The number of medic agents available
9. The number of police agents available
10. The initial starting position of the agents
11. The number of hospitals
12. The location of the hospitals
13. The number of civilians that need to be rescued
14. The location of the civilians

15. The time it takes to rescue a civilian so that they can be transported to hospital (buried-ness)

16. The deadline that each civilian must be rescued by to obtain reward

Many of these variables were kept constant for every problem: edge length, proportion of un-traversable edges to total edges; blocked-ness of un-traversable edges; initial starting position of agents; the number and location of hospitals; and the configuration of nodes. Nodes were always arranged in a square layout with an equal width and height, and were connected to their Manhattan neighbours (the nodes closest in the cardinal compass directions). This means that the number of nodes in graph is always a square number, and increases proportionally to the square of the width of the graph. Edge length was set such that it took 120 seconds to traverse and the blocked-ness of blocked edges was set such that it took 300 seconds to clear. This was done to keep the amount of time spent either travelling or clearing edges from dominating the other. The absolute numbers are arbitrary, but establish a balance between the time spent planning and the time spent acting. Buried-ness of civilians was also set at 300 seconds. The number of hospitals was set at two, with one hospital in the top left corner and one in the bottom right corner. This was to was done to create localised areas of tasks, rather than one central hub from which all tasks would revolve around. Agents all started at hospitals and were evenly distributed (by number and type) among the two. The location of civilians were uniformly selected at random from all nodes, excluding those that were hospitals.

Figure 5.2 is an example of one of the types of problems used. The grey squares and dark green squares indicate nodes – grey for ordinary nodes and dark green for hospitals. Lines between the nodes indicate edges – unblocked edges are cyan and blocked edges are red. Circles indicate agents and civilians – green is a medic, blue is a police, and olive is a civilian. The location of an agent/civilian is the node to its immediate upper left. We can see that whilst some civilians are spatially closer to one hospital it may be quicker to take the civilian to the other hospital. For instance, *civ2* is closer to *hospital0-0* (240

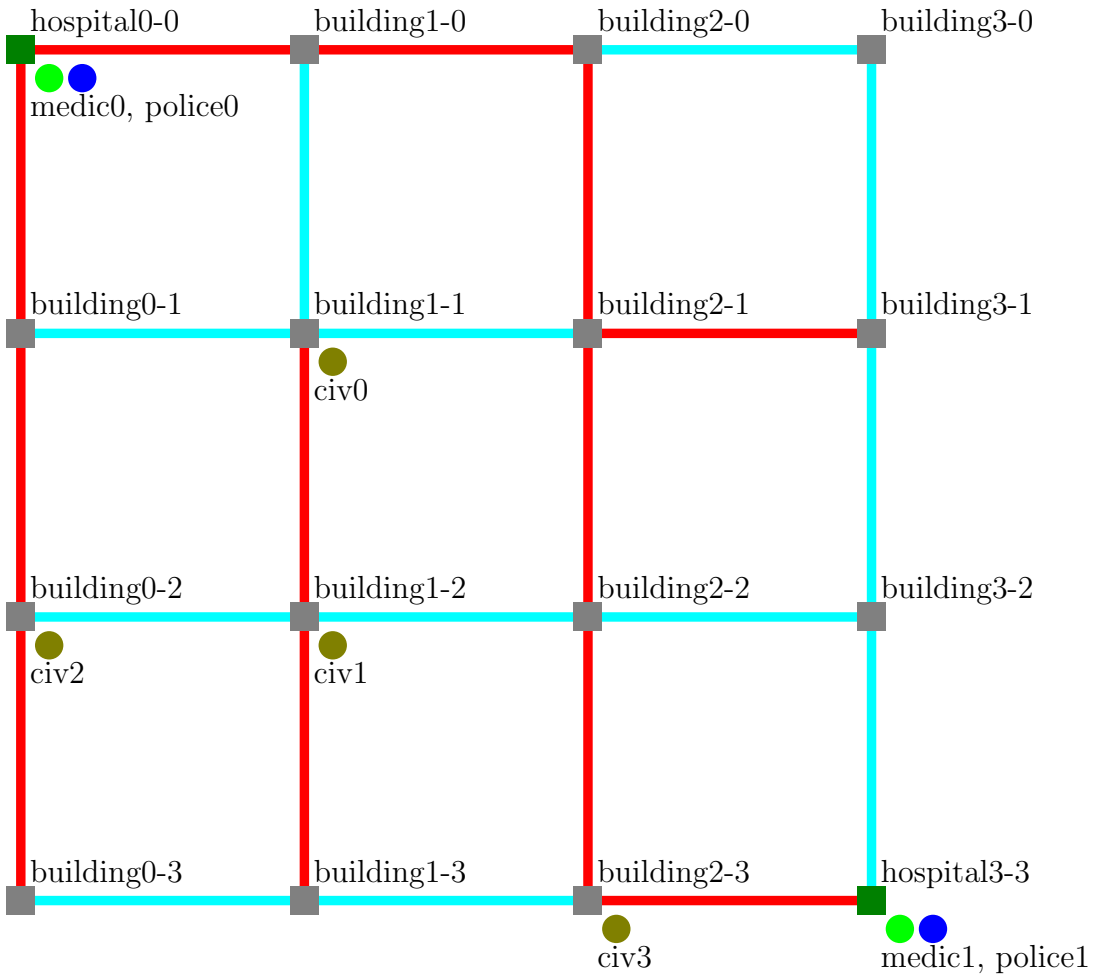

Figure 5.2: Example problem in initial state

units), but it will take a minimum of 840 seconds[1] for *medic0* to reach *civ2*. In comparison *medic1* is 480 units from *civ2*, but it will only take 480 seconds for *medic1* to get there as there is a direct route from *hospital3-3* to *building0-2* that does not require any edges to be cleared.

Some independent variables were tied to the value of other variables: heuristic time was linked to planning time; the number of blocked edges proportional to the number of edges in the graph; the number of police agents was equal to the number of medic agents; and the number of civilians that need to be rescued was a function of the number of nodes in the graph. Heuristic time was always one tenth of planning time. This is because the amount of time spent on deciding on how costly a task is for an agent to complete should be low in comparison the amount of time actually spent planning on how to complete the tasks assigned to the agent. In the initial state 50% of edges were blocked. This was done so as to ensure that there would be at least some cooperation required by police and medic agents. Edges were uniformly selected at random to be initially blocked. The number of police agents was always equal to the number of medic agents. Finally, the number of civilians to be rescued was always one quarter of the number of nodes in the graph. This was done so that scaling the problems up only requires the changing of one variable. The location of a civilian was selected uniformly at random from all nodes that were not hospitals. This means that more than one civilian can be at a given node.

The remaining independent variables are: planning time; number of nodes in the graph; number of agents; and time by which a civilian needs to rescued by. For the possible values of these variables and the variables that covary with them, please see Table 5.1 on page 97. Planning time represents the ability to find better solutions, but at the trade off of taking longer to plan. A longer planning time, will lead to a better quality of solution for the start time that was planned for, but the later start time that results from a longer planning time may lead to lower overall reward achieved. The number of nodes in

[1] 840 seconds comes from the police agent having to clear one edge (300 seconds) and then move to *building0-1* (120 seconds) and clear the next edge. The medic can travel to *building0-1* at the same time as the police and then travel to *building0-2* once the second edge is cleared ($840 = 300 + 120 + 300 + 120$).

the graph serves to increase the difficulty of the problem. The number of nodes can only increase in square increments (eg. $1, 4, 9, ..., n^2$), making successive problems quadratically harder than the last. Finally, the number of agents in the problem also effects difficulty, but the relationship is expected to be inversely proportional to the number of agents. The more agents there are, the better overall solution that is produced. Conversely, in a centralised approach more agents could lead to less reward being achieved. This is because an increase in the number of agents means an increase in the branching factor of the problem. This means the planner must consider a larger state space, and will, as a result, take longer to find higher reward plans. In continual domains (like RR), this will lead to agents being slower to achieve goals and so reducing overall reward. However, due to a decentralised approach, adding additional agents will not increase planning complexity. This is because each agent never directly considers other agents[1]. More agents should also lead to a lower average number of tasks assigned to each agent. With less tasks to plan for, it should be easier to compute plans that achieve all tasks (as planning time is fixed) and have a lower makespan – thus achieving a higher global reward.

We tested our algorithm on two types of problem. One with deadlines and one without. In problems without deadlines, the quality of solution was based on the time it took to rescue all civilians. To create problems with deadlines, the previous problems without deadlines had deadlines added. For each problem instance, deadlines were set based on the time it took to rescue all civilians in the respective non-deadline problem instance. The deadline by which to rescue a given civilian was selected from a uniformly random distribution between zero the completion time on the non-deadline problem instance. This will have the effect of causing the problem to not be fully solvable in some circumstances, but also means a similar proportion of civilians should be 'rescue-able' in all problem instances. That is, problems where it took to rescue all civilians get longer to save civilians. By looking at how successful the algorithm is when under time pressure, we can see how well the algorithm is at organising cooperation between agents.

[1]Agents indirectly consider other agents when achieving subtasks.

Variable	Values
Width of Graph Layout	4, 5, 6, ..., 19, 20
Number of Nodes	16, 25, 36, ..., 361, 400
No. of Civilians	4, 6, 9, ..., 90, 100
No. of Medics	2, 4, 6, 8, 10
No. of Police	2, 4, 6, 8, 10
Task deadline	No, Yes
Planning Time	10, 60, 100
Heuristic Time	1, 6, 10

Table 5.1: Possible values of variables, with co-varying variables grouped.

5.4.2 Dependent Variables

The dependant variables that are used to measure the performance of the algorithm are:

1. Time spent allocating tasks to agents
2. Time spent when there is at least one agent planning (planning time makespan)
3. Total time spent by all agents planning (total planning time)
4. Completion time
5. Percentage of civilians rescued

The percentage of civilians rescued is the main metric by which to measure the quality of a plan. A plan that achieves more of its goals is preferable to one that achieves less. This is especially true of the RR problem.

Completion time is a second metric that can be used to measure the quality of the plans that are produced. This is why the problems where all civilians can be rescued exist. Since all civilians can be rescued in the problems without deadlines, the only metric to measure the quality of plans is the makespan of the plan. A plan that achieves all the given goals before another plan is better.

Time spent allocating is expected to linearly increase with the number of tasks that need to be completed. That is, the police agents plan, then the medic agents plan. If task allocation is unable to allocate all tasks because agents were unable to come up with cost

estimates given the heuristic time for the problem, then remaining tasks will be allocated once all currently allocated tasks have been completed. Agents will cycle between task allocation, planning, and execution until all tasks have either been achieved or are no longer achievable (as their deadlines have passed). If the algorithm only plans once then this indicates that the algorithm is successfully de-constructing the problem into smaller problems to be solved. Whereas, if planning is required multiple times then the algorithm has been unable to fully de-construct the problem instance in one pass.

5.4.3 The Effect of Changing Planning Time

This section examines how changing planning time effects plan quality. In these problems there was no time pressure to complete tasks (no task had a deadline by which it had to be completed). Thus all problems could be solved given sufficient time and planning resources. As such, the only metric to used to establish the quality of the plan was how long it took for all tasks to be completed. As problems got sufficiently complicated, plans could no longer be produced given planning resources. All problems only had two of each type of agent (4 overall).

Figure 5.3 shows how long was spent allocating tasks to the agents. This time is the makespan of the time taken by agents to compute the cost estimates (bid values) for each task. The figure shows how total allocation time increases with grid height. This is because of an increase in number of tasks (civilians to be rescued). Recall that the number of civilians to be rescued (tasks) is tied to the number of nodes in the graph, and the number of nodes in the graph is the square of the grid height.

Figure 5.3 shows how allocation time varies for each type of planning time as the grid height increases. This figure shows that total allocation time increases with both problem complexity and with the amount of time each agent spends coming up with a bid for each goal (allocation time). It shows that the proportion of total time spent allocating remains proportional to allocation time regardless of problem complexity. This shows that increasing allocation time does not necessarily lead to a better distribution of goals

amongst that agents. That is, with a larger problem, increasing allocation time should be less than linearly proportional to total allocation time. Meaning better initial goal allocations mean more goals can be achieved faster, and thus leading to less goals having to be allocated in future goal allocations. Figures 5.4 to 5.6 show the same data as in Figure 5.3, but individually for each allocated planning time. Figures 5.4 to 5.6 make it clear that time spent allocating tasks increases linearly with the number of number of tasks, until such a point that the algorithm cannot fully solve the problem instance. For instance, in Figure 5.4 the time spent allocating tasks is linearly proportional to the number of tasks for the first six data points. The subsequent three data points are where the algorithm cannot achieve all the tasks as the as agents cannot compute bid values in the allotted time. As such, the time spent allocating tasks is not proportional to the number of tasks in the problem instance.

After a certain point, allocation time stops increasing (in Figures 5.3 to 5.6), this is where the problem has become sufficiently complex that the algorithm cannot fully solve the problem. As problem complexity increases the time given for calculating the cost heuristic is no longer sufficient to compute relaxed plans that achieve the task. As such, the algorithm is unable to allocate those tasks. This also means that the algorithm was unable to achieve all the tasks in the problem. Figure 5.7 shows allocation time against the number of civilians rescued. There are a number of problems where the algorithm was unable to complete a single task. Aside from these problems, there is a rough correlation between the number of civilians rescued and time spent allocating tasks.

Figures 5.8 and 5.9 show planning times versus grid height. Total planning time is the sum of the time spent planning by all agents over the course of the problem. Planning makespan is the total time where at least one agent was planning. Thus a situation where one agent plans from t_0 to t_1 and another plans from t_0 to t_2 would have a total planning time of $((t_2 - t_0) + (t_1 - t_0))$, and a makespan of $t_2 - t_0$. These planning times are for problems with 2 agents of each type, thus total planning time is usually twice that of the planning time makespan. This demonstrates that there are normally only two

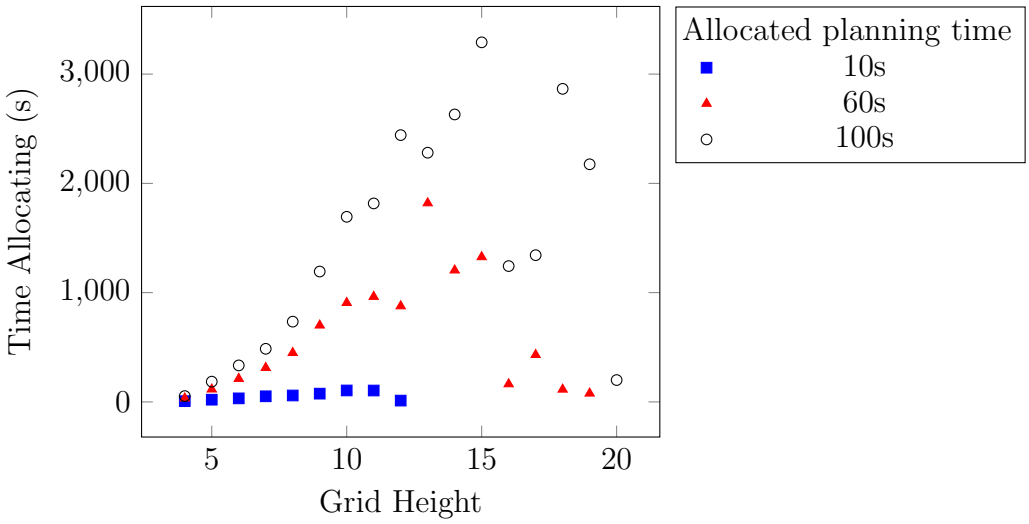

Figure 5.3: Grid Height against Allocation Time

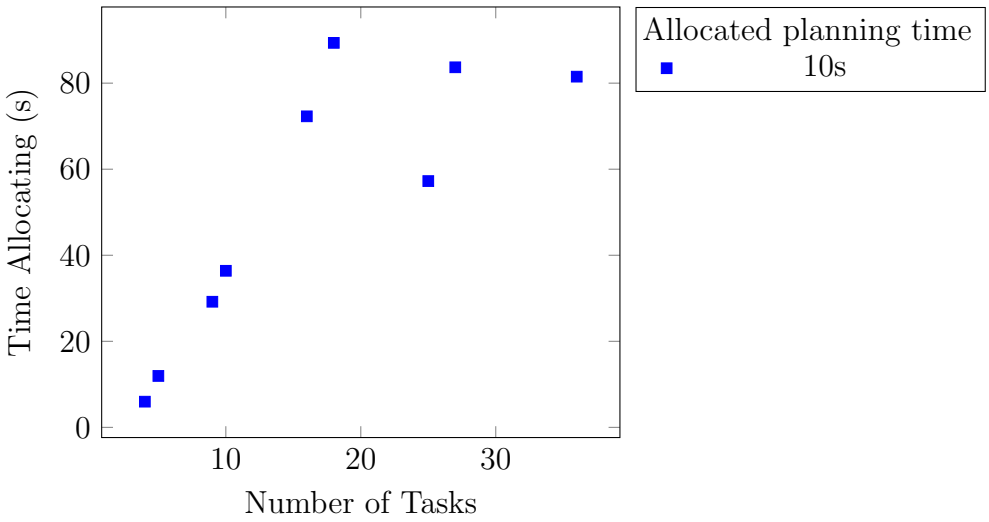

Figure 5.4: Number of Tasks against Allocation Time

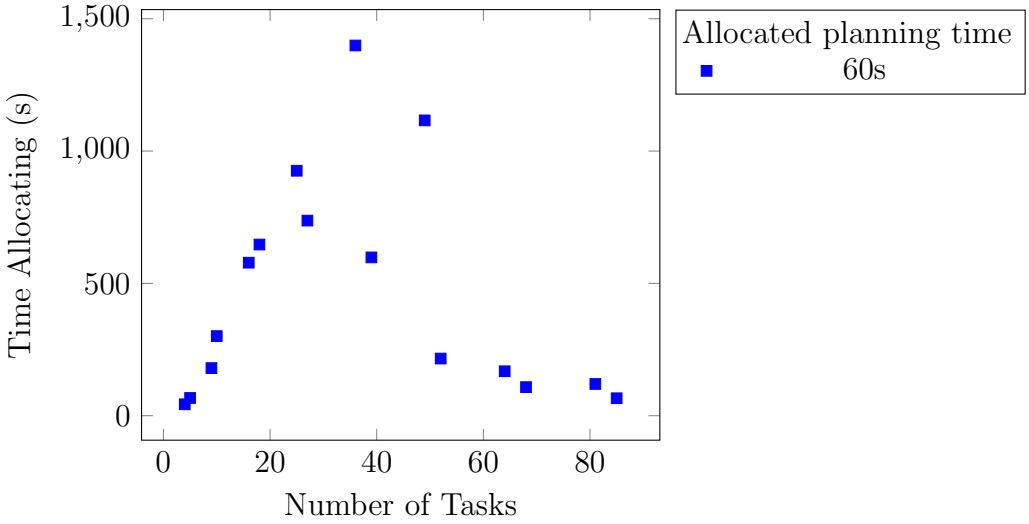

Figure 5.5: Number of Tasks against Allocation Time

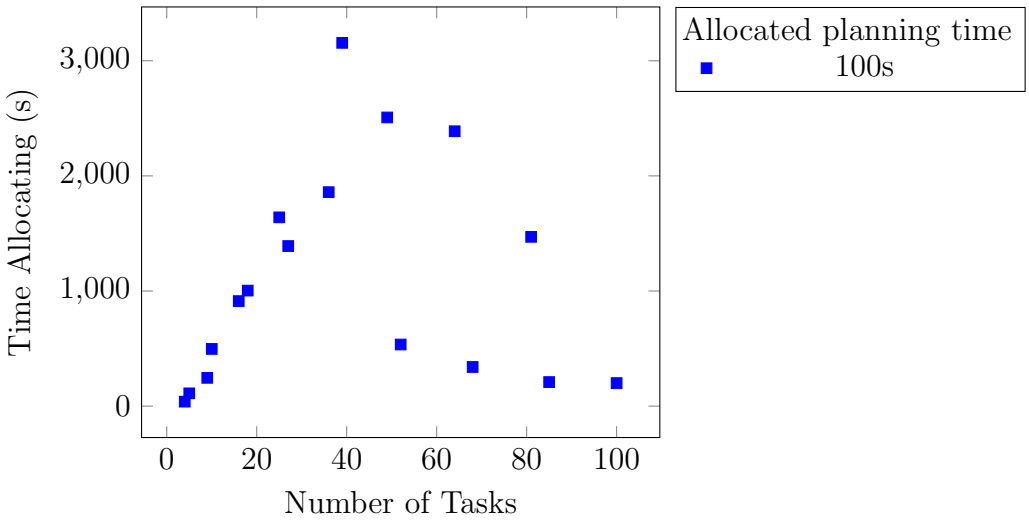

Figure 5.6: Number of Tasks against Allocation Time

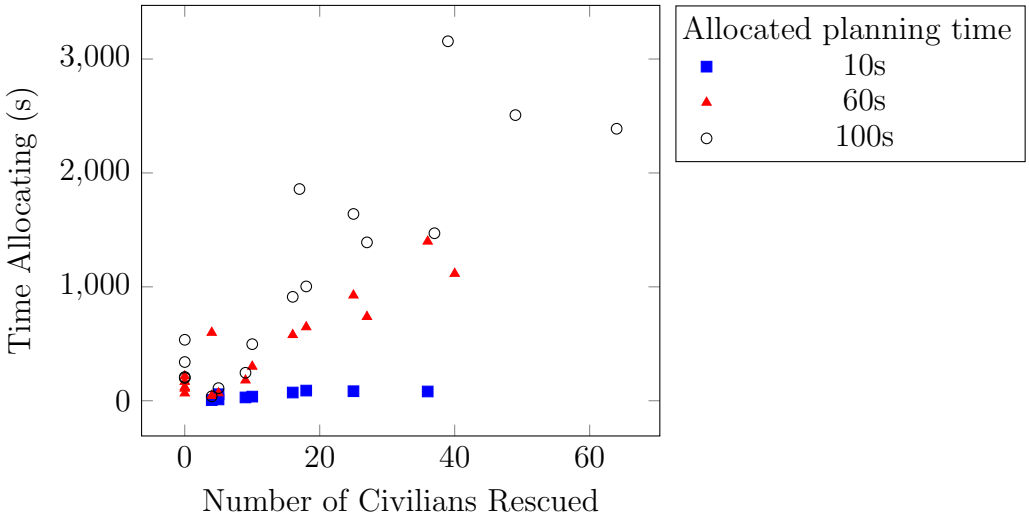

Figure 5.7: Civilians Rescued against Allocation Time

agents planning at the same time. Theoretically this makes sense as the two medic agents cannot plan until the two police agents have finished planning and reported when they will complete the medics' subtasks by.

Typically these planning times are just multiples of time that agents had to spend planning for their allocated tasks. They represent the number of times task allocation occurred. The first four problems (with grid heights 4 through 7), all only allocate tasks once, and thus only plan once. Subsequent problems have larger planning times, this is due to the algorithm being unable to completely allocate all tasks the first time round. Each agent is allowed to complete the plan it computes for its task before a second round of tasks allocation occurs.

For very simple problems, however, planning may find the optimum solution before planning time has fully elapsed. This can be seen in Figure 5.8 where total planning time for the 4×4 grid when planning time was 100 seconds. This indicates that one of the agents was able to find the optimal plan in the time allotted and terminate planning early.

If a round of task assignment and subsequent planning occurs where no agent is able to produce a plan that completes any of the tasks assigned to it, then the algorithm stops. This happens for more complex problems, and shows that there are still outstanding tasks, but that they are unsolvable given current resources. This could be because there exists

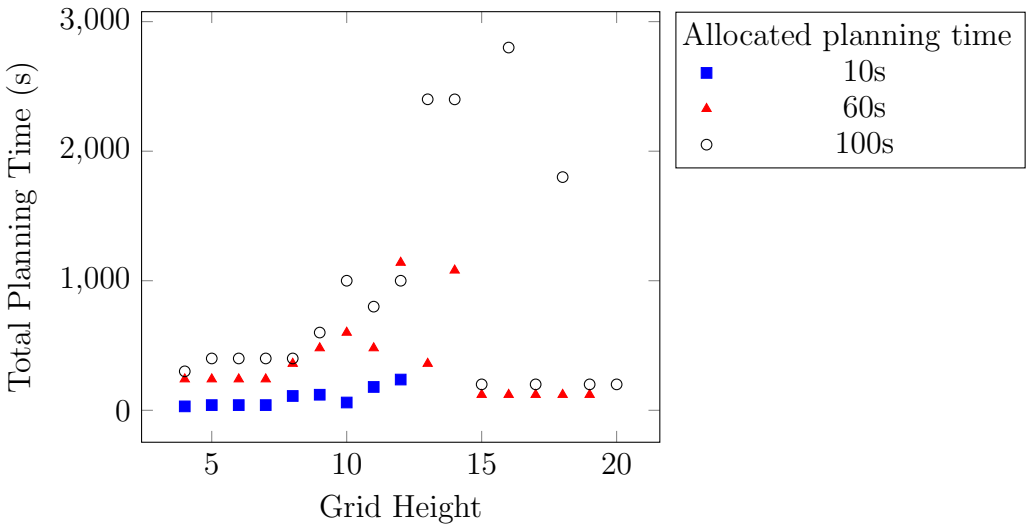

Figure 5.8: Grid Height against Total Planning Time

no possible solution for the given goal, or that no possible solution can be computed given current computational restrictions on planning time or heuristic time.

Figure 5.10 shows how completion time varies with problem difficulty, across various different planning times. It shows that typically, shorter planning times are able to produce similar results right up until they are unable to produce any plans whatsoever. For the purposes of gaining additional reward on a specific problem, increases in planning time do not sufficiently decrease plan makespans to justify such an increase.

Figure 5.11 shows the proportion of tasks completed. This bar graph makes explicit what can be deduced from Figures 5.8 and 5.9 (that the main effect of increasing planning time is that harder problems can be solved). It shows that the ability to complete a problem drops off erratically. This could be the result of the random distribution of blocked roads in the graph. That is, police agents can have considerably different problem difficulties, and that this difficulty depends on the layout of civilians and blocked roads (rather than just grid size).

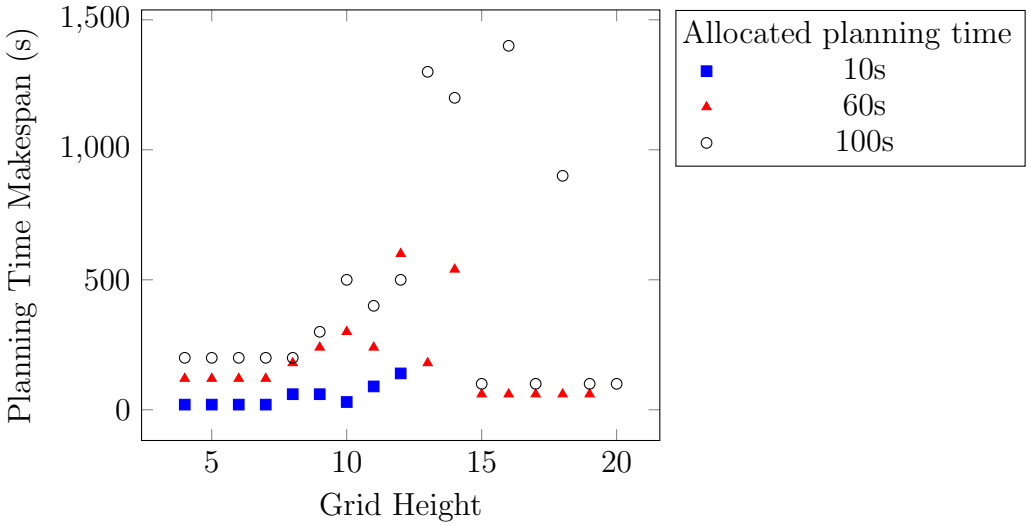

Figure 5.9: Grid Height against Planning Time Makespan

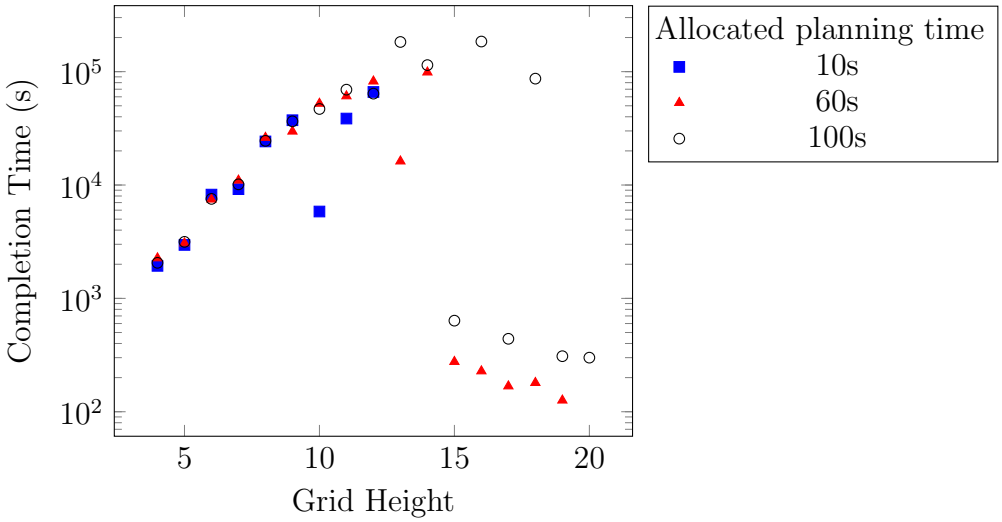

Figure 5.10: Grid Height against Completion Time

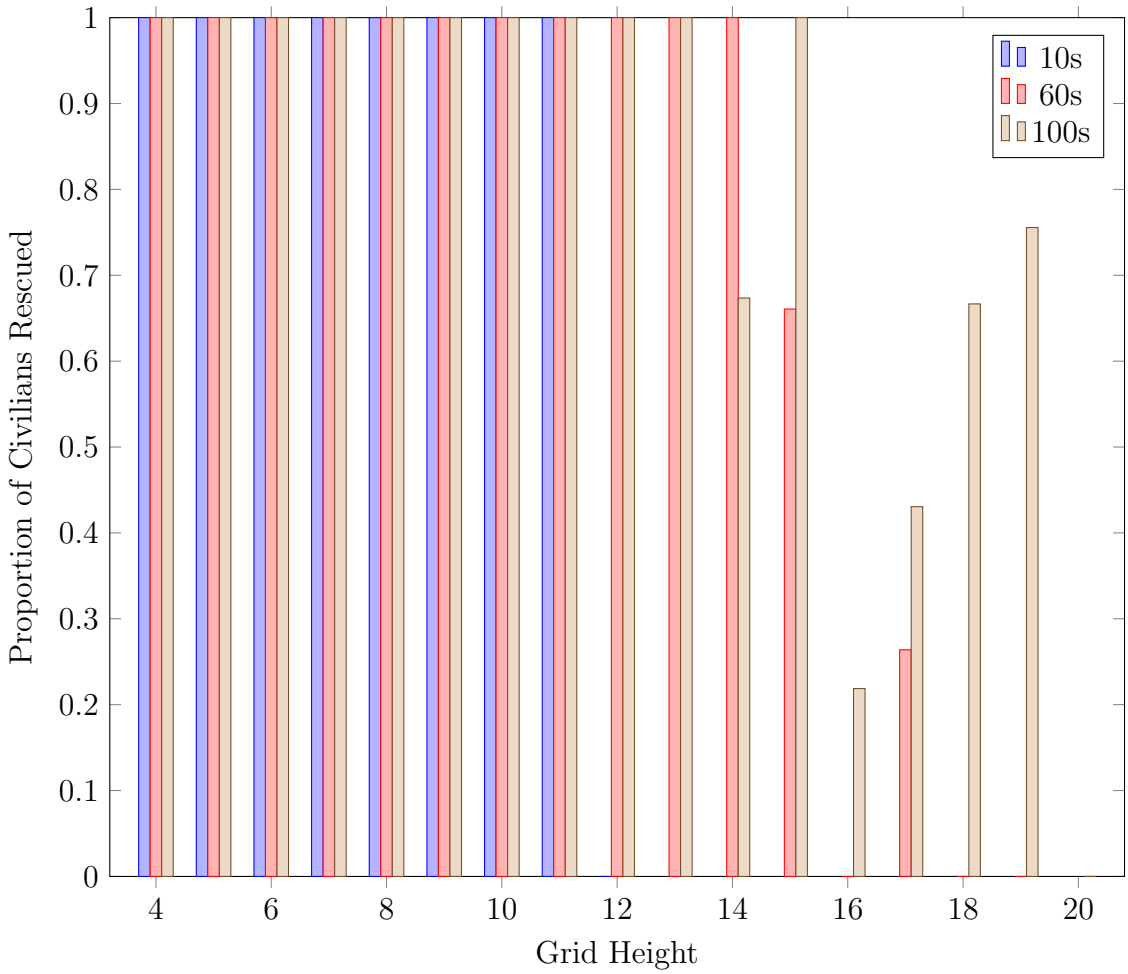

Figure 5.11: Grid Height against Proportion of Civilians Rescued, without deadlines – $civilians = (\frac{height}{2})^2$

5.4.4 The Effect of Time Pressure

This section discusses the effect of deadlines on the problem domain, and how this effects the performance of the algorithm. The primary comparison will be the proportion of civilians rescued. However, the time spent allocating tasks to agents, the time spent planning and the completion time will also be looked at. In these problems there are two agents of each type (4 overall).

For each problem deadlines were added to each task to create a new problem. All other parts of the problem remained the same. This has the effect of making the problem oversubscribed. The deadline for each task was the time by which it must be achieved to obtain reward. The agent either gains all the reward for rescuing a civilian or none at all. There was no reward gained for rescuing civilians sooner rather than later. The deadlines were taken from a uniform distribution between zero and a maximum value. The maximum value was the completion time of the algorithm on the same problem without deadlines, when given 60 seconds planning time and 10 seconds allocation heuristic time (see Figure 5.10). The mean average deadline for rescuing an agent was therefore half the completion time.

Figure 5.12 shows the proportion of civilians rescued for each problem. It shows that where the algorithm was able to successfully rescue civilians, it managed to rescue 50% or more in all but two cases. In the cases where 50% or more were rescued the average was 70%. As the number of civilians increases, so too does the proportion of civilians rescued. The lower proportion of civilians rescued in simpler problem instances can be explained by the fact that the algorithm attempts tasks with earlier deadlines first, and that by doing so it precludes the algorithm from achieving a higher proportion of civilians rescued than it otherwise could have.

The time required to allocate tasks remains proportional to planning time. This is shown by Figure 5.13 in comparison to Figure 5.3. By looking at the allocation times

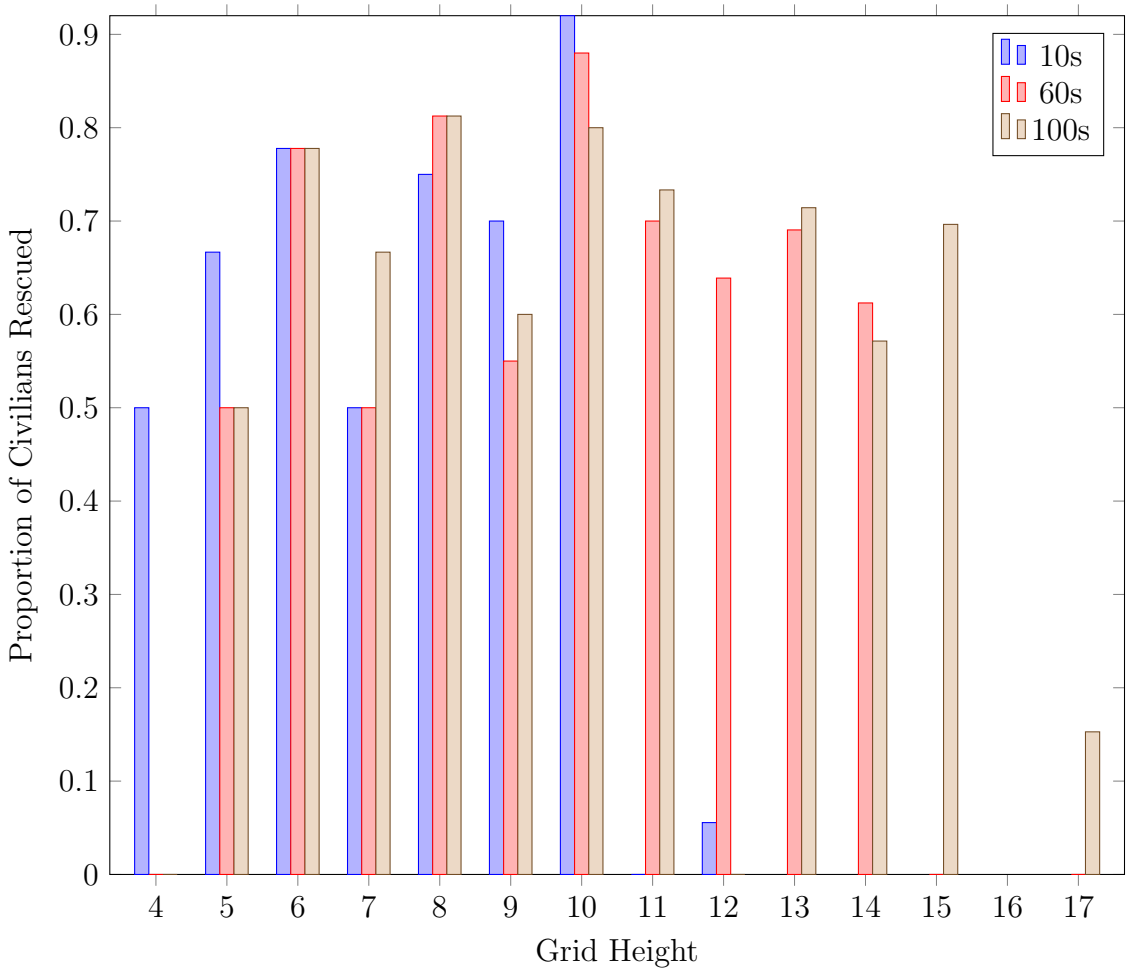

Figure 5.12: Grid Height against Proportion of Civilians Rescued, with deadlines – $civilians = (\frac{height}{2})^2$

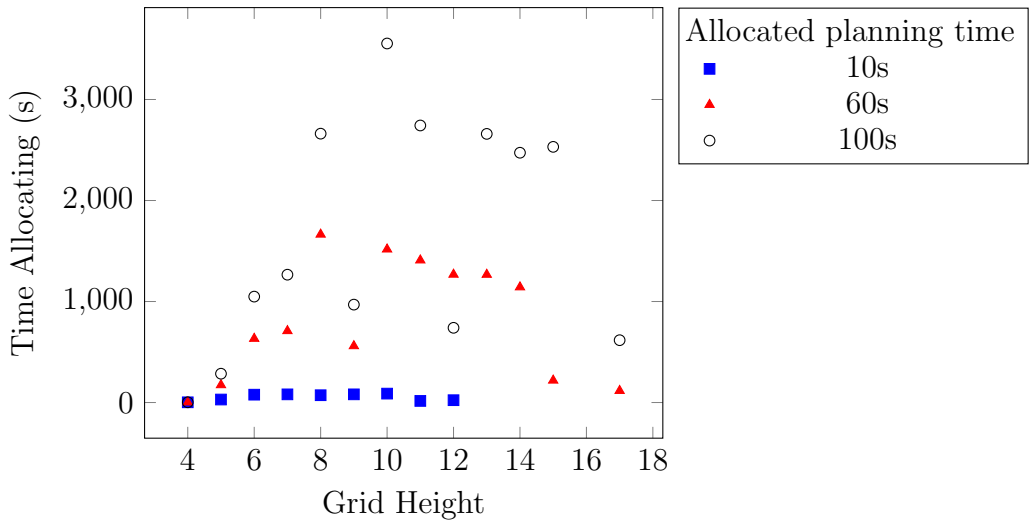

Figure 5.13: Grid Height against Allocation Time when domain has Deadlines

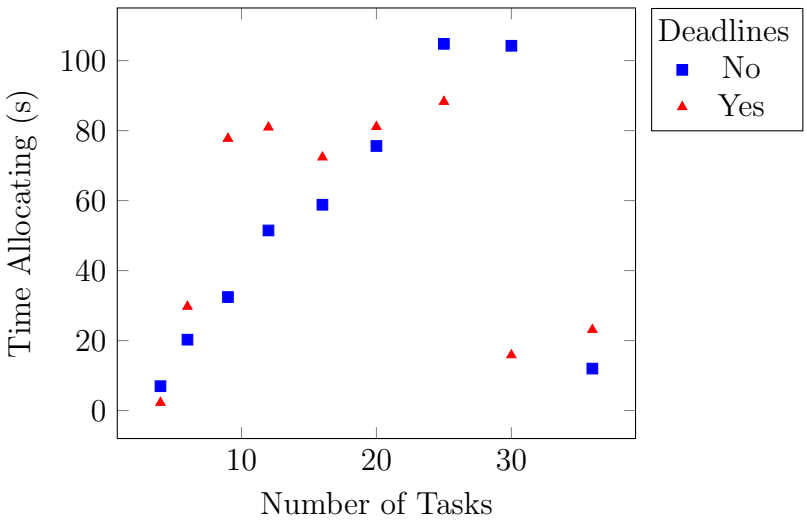

Figure 5.14: Number of Tasks against Allocation Time when Planning Time is 10s, comparing domains with and without Deadlines

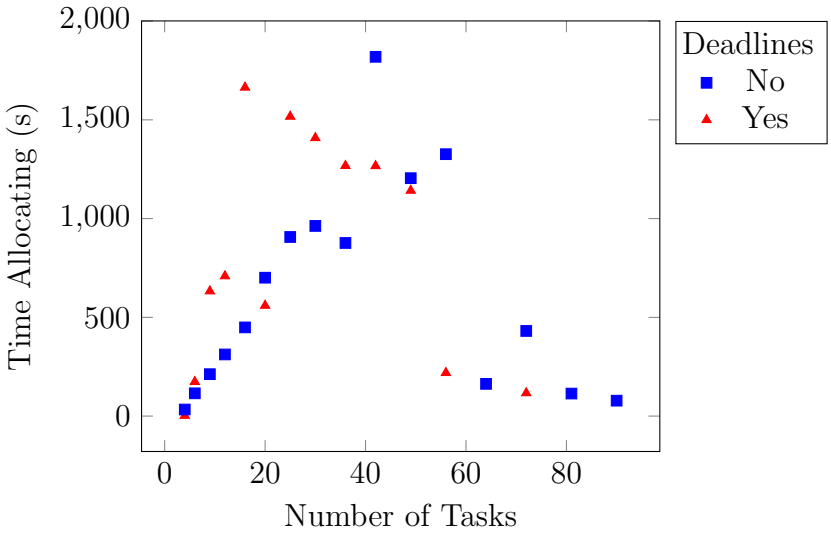

Figure 5.15: Number of Tasks against Allocation Time when Planning Time is 60s, comparing domains with and without Deadlines

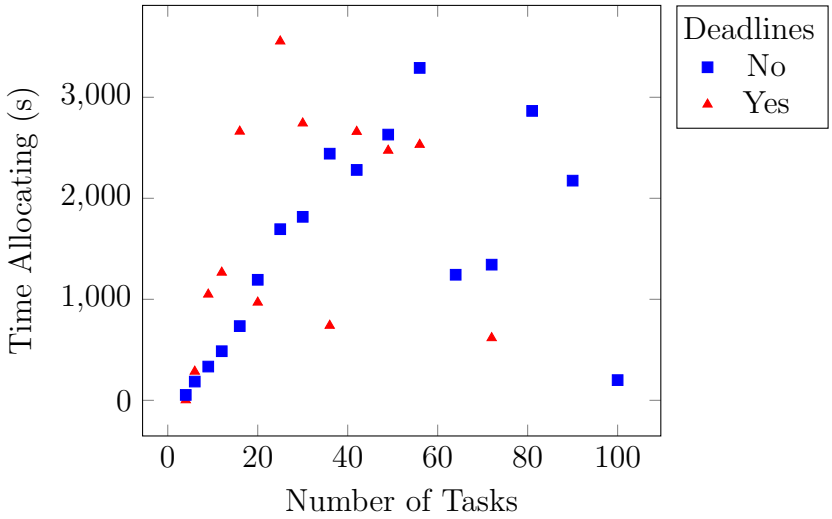

Figure 5.16: Number of Tasks against Allocation Time when Planning Time is 100s, comparing domains with and without Deadlines

for separate planning times we are able to see the impact of deadlines on the algorithm. Figure 5.14, Figure 5.15 and Figure 5.16 show that allocation takes longer but becomes less proportional to problem complexity (grid height). This could be explained by that fact that the tasks are allocated according to their deadline (those with the closest deadline allocated first). However, this does not take into account geographical proximity of tasks and agents may change the order in which they attempt tasks when coming up with their heuristic cost. As such multiple subtasks that achieve the requirements of a single task in different ways could be issued. That is, a medic agent might request two routes to a civilian to be cleared when it only needs one. This is because the task allocation algorithm contains no way to recall subtasks that are not deemed necessary any more.

5.4.5 The Effect of Varying the Number of Agents

In this section the effect of an increased number of agents on the performance of the algorithm is examined. This allows for a secondary examination of how the algorithm performs under increasing complexity. The first measure of complexity was grid height. The number of nodes in the graph and number of civilians in the problem is always proportional to grid height. By increasing the number of agents in the problem the

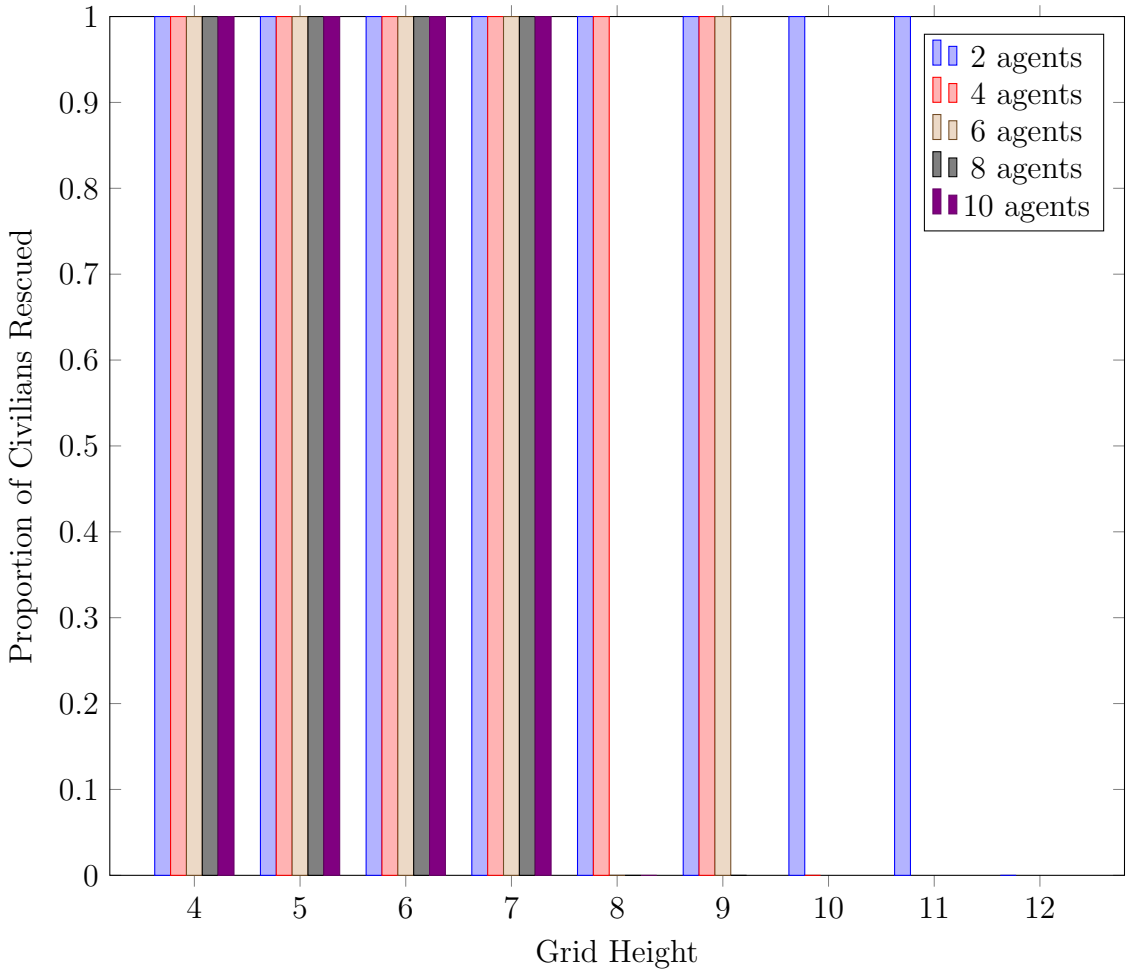

Figure 5.17: Grid Height against Proportion of Civilians Rescued

branching factor of search is increased. Whereas, increasing the grid size increases the length of the plan required to solve a problem. In these problems there are no deadlines.

First it is shown that the algorithm is able to cope with an increased number of agents, but this comes with a reduced ability to deal with larger grid heights (see Figures 5.17 to 5.19. This is surprising as the objective was to decouple the complexity of problems that can be solved from the number of agents in the problem. Figures 5.20 to 5.22 show that the algorithm is able to take advantage of the number of agents in the problem by significantly reducing the time it takes to complete the goal of the problem (completion time).

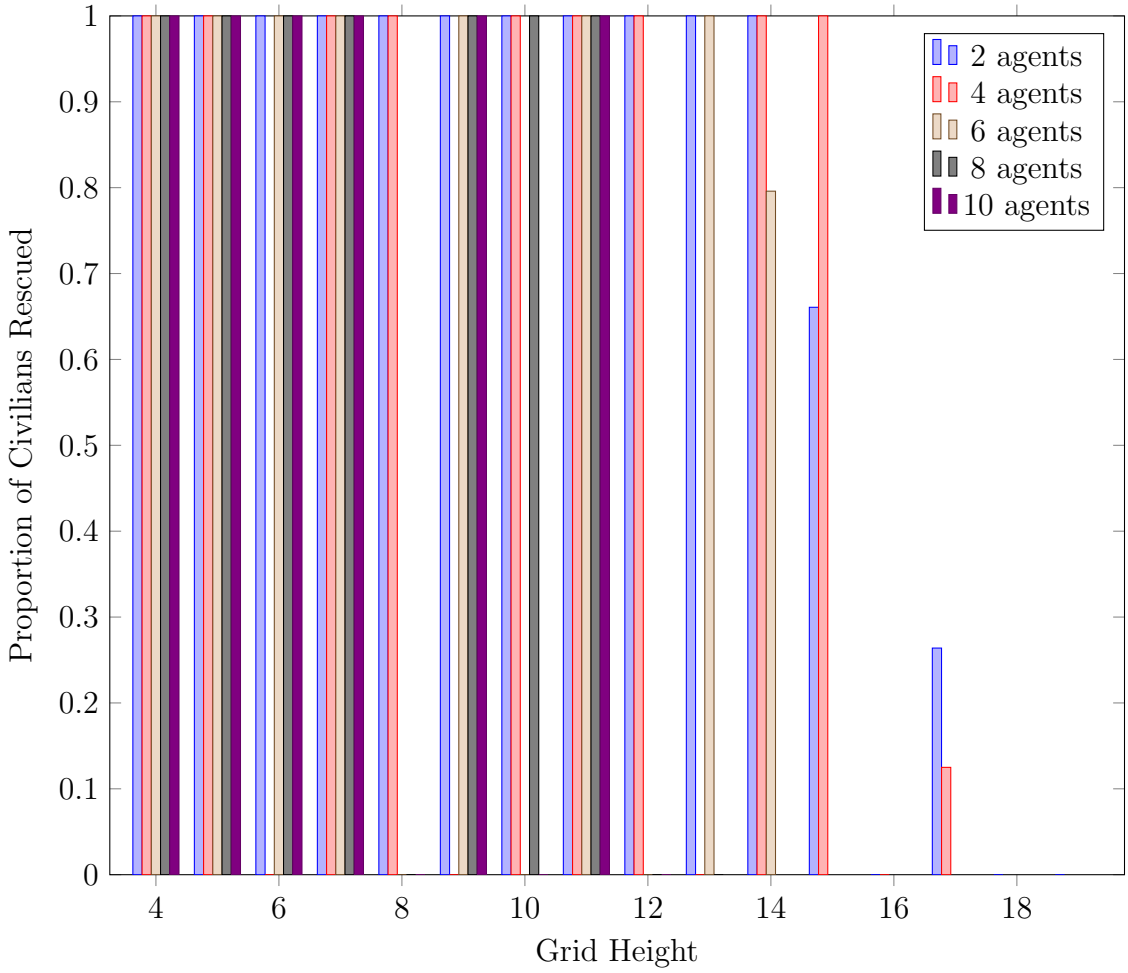

Figure 5.18: Grid Height against Proportion of Civilians Rescued

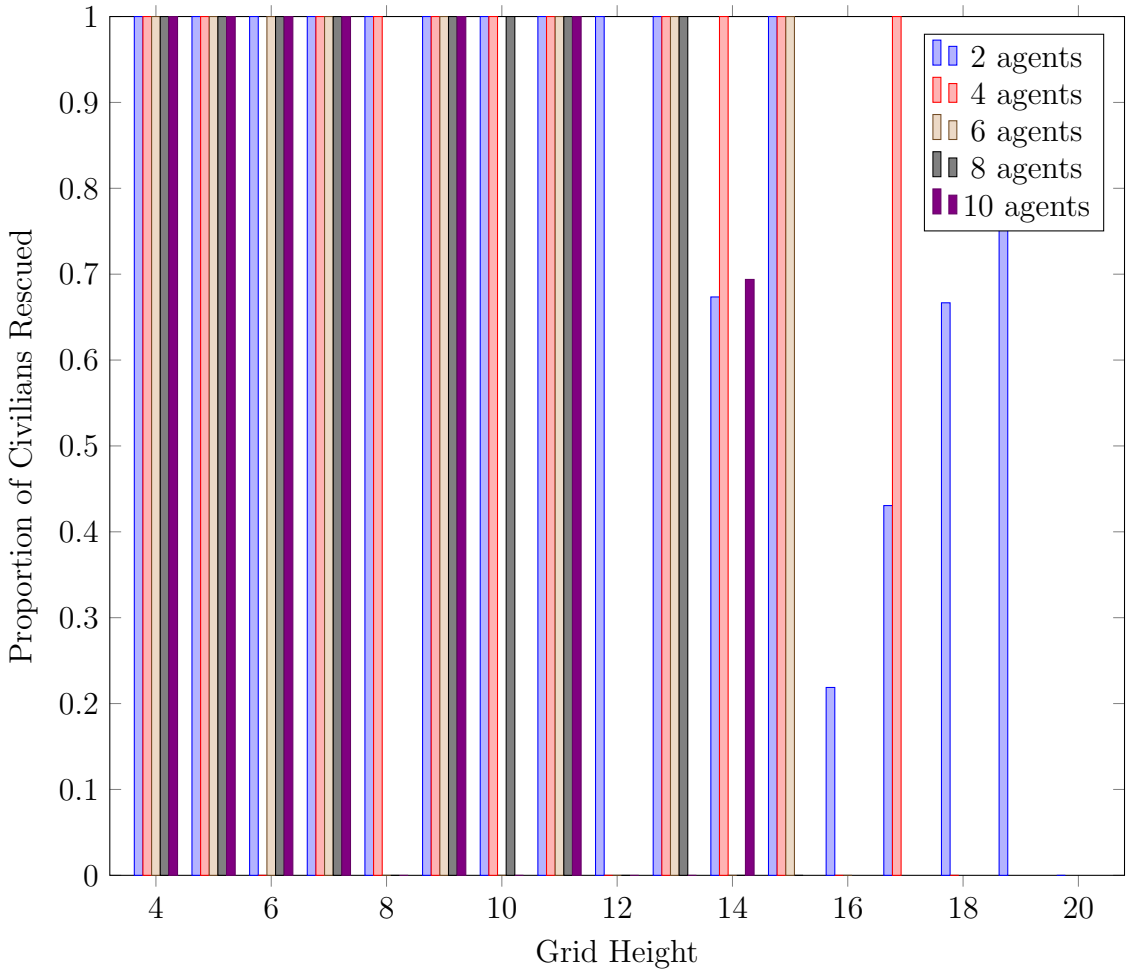

Figure 5.19: Grid Height against Proportion of Civilians Rescued

5.4.6 Comparison to Centralised Planner

This section demonstrates the quality of the multi-agent approach the algorithm takes in comparison to a centralised planner that considered the problem as a whole. The centralised planner was tested on problems without deadlines, 2 agents of each type, and all grid sizes. The planner uses the same off-the-shelf planner that the multi-agent algorithm uses. The planner was given 6 hours to compute a solution for each problem. As such the centralised planner was not tested in domain with a shared time budget

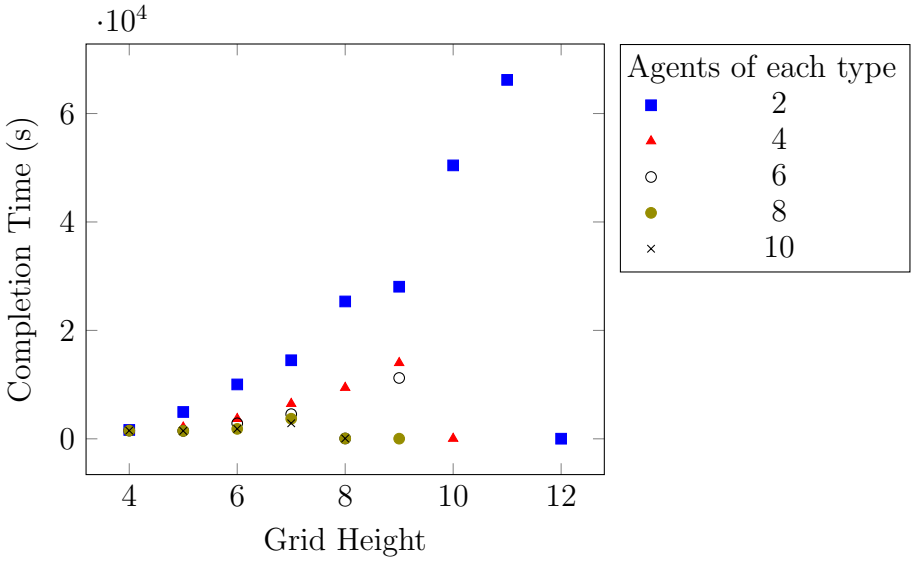

Figure 5.20: Grid Height against Completion Time, with planning time of 10 seconds

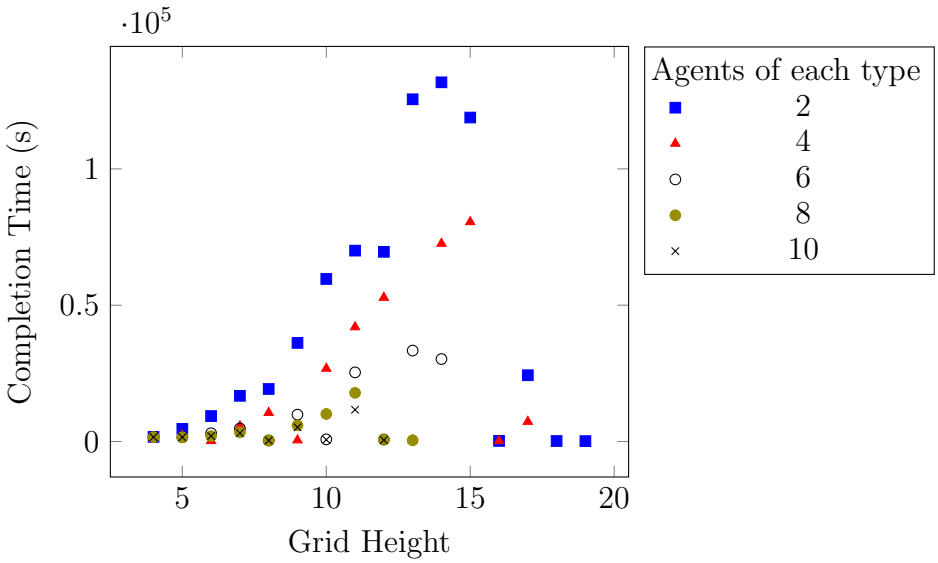

Figure 5.21: Grid Height against Completion Time, with planning time of 60 seconds

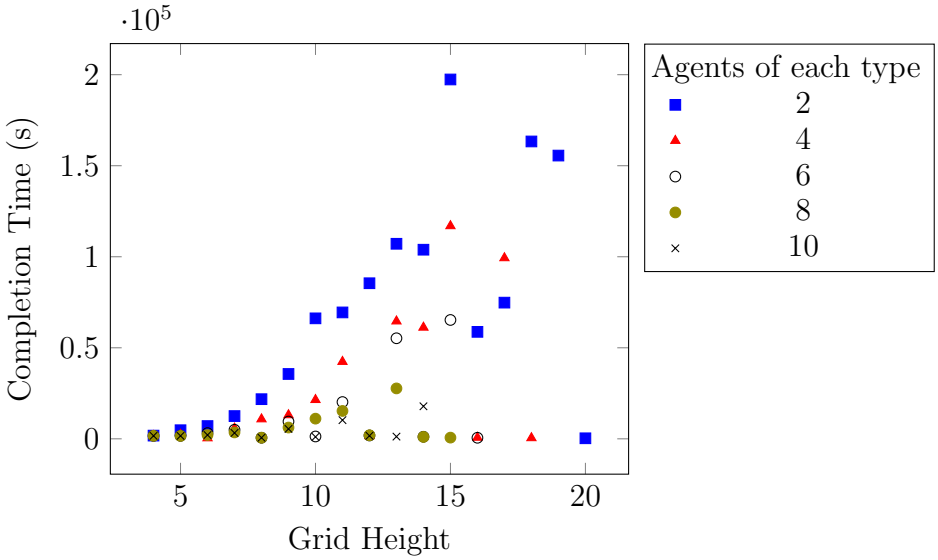

Figure 5.22: Grid Height against Completion Time, with planning time of 100 seconds

between planning and execution. Rather only planning was done, and the resulting plan assumed to have been executed from time zero.

The first thing to note is that the centralised planner is only able to produce plans for the 4 smallest grid heights, and only able to completely achieve all reward for the problem with the smallest grid height (as shown in Figure 5.23). In comparison the multi-agent algorithm with a 10 second planning time (per agent) is able to fully solve the first 8 problems (grid heights 4 though 11 inclusive). It should be reiterated that the complexity of the problem is quadratically proportional the grid height of the problem. This shows that the multi-agent algorithm is able to solve significantly harder problems than the centralised planner with only a fraction of the computational resources. In comparison to the 21,600 seconds that the centralised planner had, the multi-agent algorithm returned a solution after a combined total of 30.3 seconds. When the ability of the algorithm to parallelise computation is taken into account that time is reduced to 19.9 seconds – 0.23% of the time the centralised planner took. Finally, not only does the multi-agent solution compute a solution faster, it also produced a better solution. The execution makespan for the centralised planner was 1980 seconds, whilst the multi-agent algorithm had an execution makespan of 1620 seconds.

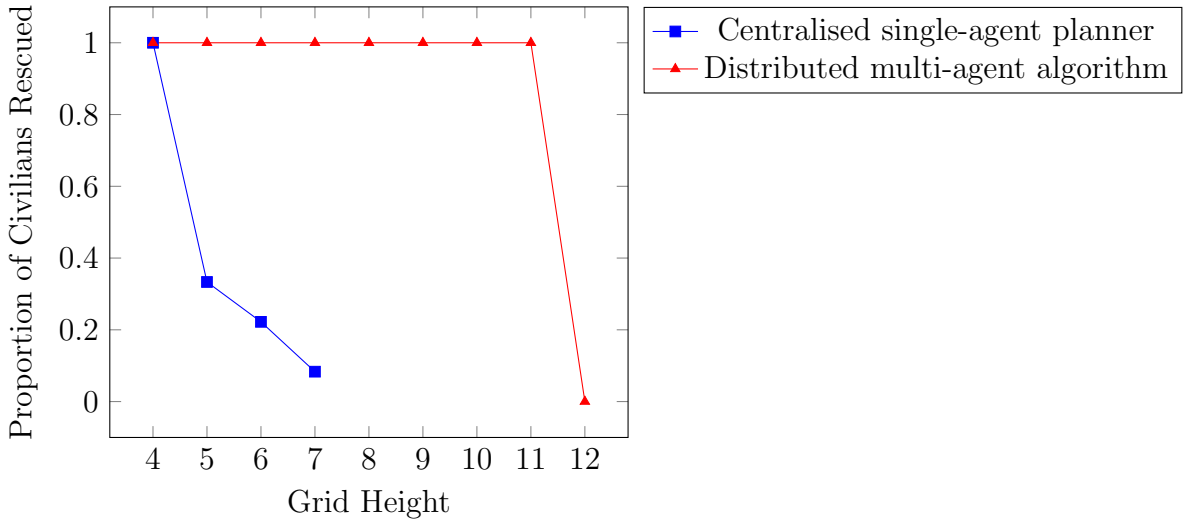

Figure 5.23: Comparison of the Proportion of Civilians Rescued for a Centralised Single-agent Approach Versus a Distributed Multi-agent Approach

5.5 Discussion

The heuristic used to see if an agent can complete a task on its own sometimes misidentified an agent as being unable to complete as task on its own. It makes no attempt to compute if other plans exist that do not require cooperation from other agents. As such, an agent may issue subtasks that are not strictly necessary. For instance, a medic agent might request a blocked road be cleared when a longer but unblocked route already exists. Further, when it comes to actual planning the unblocked route may instead be used and so the subtask was never actually required. This means agents will do more work than is necessary.

Agents are not required to complete all tasks that are assigned to them. This means that:

1. An agent may no longer require a subtask that was previously issued, because its plan does not achieve the task that was the cause of the subtask.

2. An agent may not be able to achieve all subtasks assigned to it – meaning an agent that issued the subtask may be left unable to complete all its tasks.

Some times not all of the subtasks for a particular task may be achieved. As above, this

may have the problem of meaning that the task is not achievable, but that the algorithm still thinks valuable work has been done. This is because each precondition is issued as a separate subtask rather than a collection of preconditions grouped together as one subtask. This was to: allow large groups of subtasks to be distributed amongst multiple agents; and prevent multiple agents from attempting to achieve the same precondition that was common to multiple subtasks. That is, two tasks may require the same precondition to be achieved, but the precondition only needs to be achieved once. The down side is that agents see subtasks as atomic rather than part of a larger collection of requirements. This means agents cannot reason about whether one particular subtask in a set of subtasks has no value if completed if other subtasks in the set are not also completed. For instance, consider a cross roads. If all roads and the cross roads itself are blocked then there is a value for unblocking the cross road itself, but less value for the roads leading to the cross road. This could lead to a police agent unblocking the cross road, and then moving on to higher valued subtasks rather than unblocking at least one set of roads that would make unblocking the cross roads useful (see Figures 5.24 and 5.25). As cooperation between agents is loosely coupled it is also possible for subtasks to be completed, but the relevant tasks that rely on these tasks to not be completed – meaning no actual reward is gained for the work that is done.

In the case of the problem where the grid height is four, the 60 second and 100 second planning times unexpectedly fail to rescue any civilians. This is due to the decoupling of planning between agents and decomposition of subtasks into separate tasks. For instance, if an agent requires two subtasks to be completed to achieve its goal, then it is only useful if both subtasks are completed. The way in which tasks are valued does not allow agents to take this into consideration. That is, the agent perceives each subtask as independent. The problem becomes even worse when a subtask is distributed amongst several agents. Without inter-agent coordination on which subtasks are to be achieved then there is no way to consider reward dependencies of subtasks. This is similar to the problem about cooperation discussed at the beginning of this section, but with the addition of deadlines

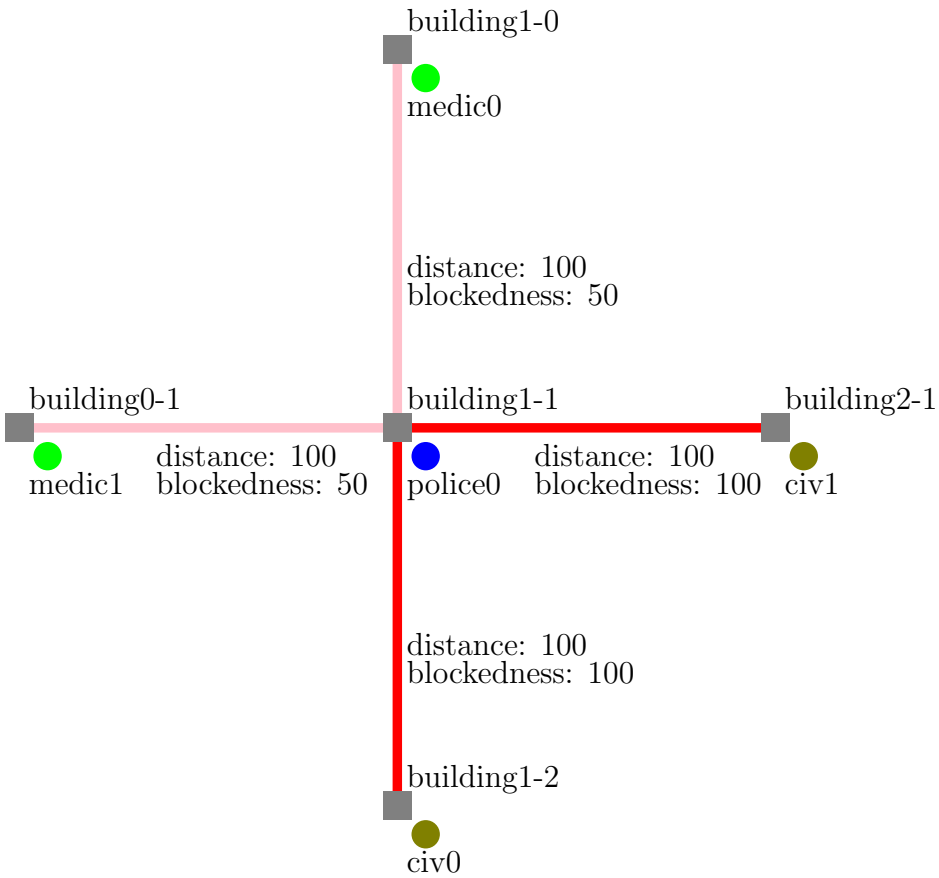

Figure 5.24: Task selection problem with tasks with varying difficulty

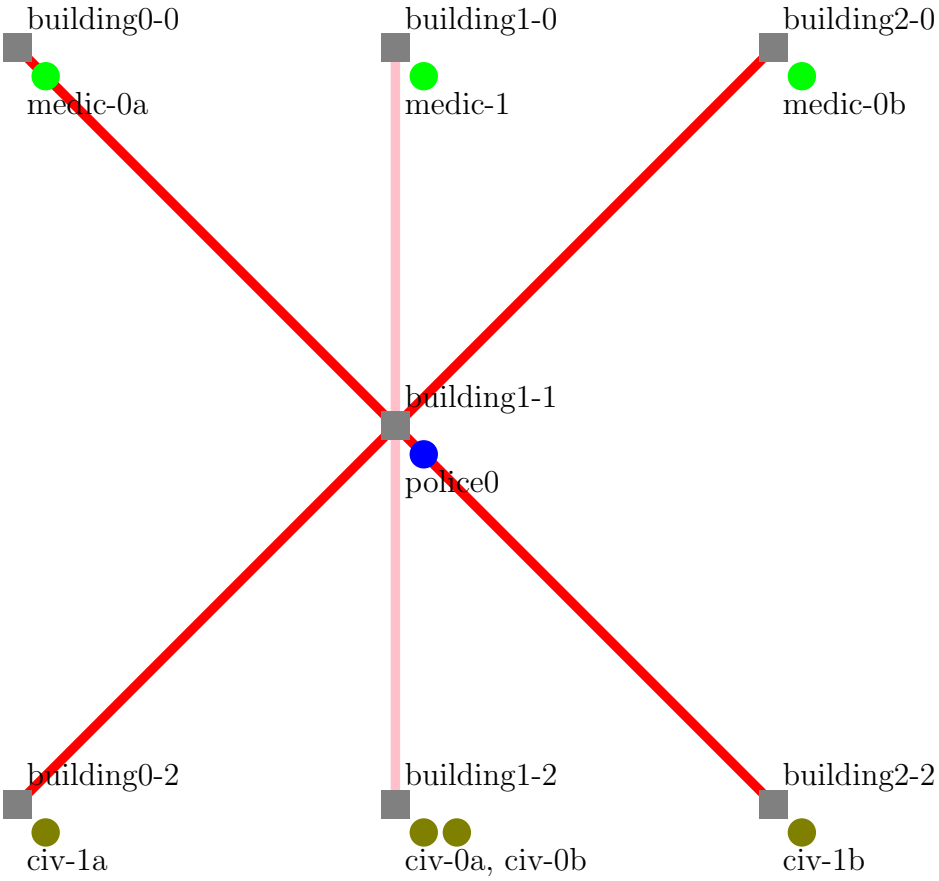

Figure 5.25: Task selection problem with similar multiple tasks issued by different agents

that induce failure rather than delays.

Separating the subtask of a task into individual subtasks was meant to be enable multiple agents to cooperate on the same subtask. That is, if an agent required two preconditions to be achieved for it to complete an action then the preconditions could be allocated as separate subtasks. This would allow for a more flexible distribution of these subtasks amongst agents capable of achieving the required preconditions. However, this makes the coupling between the task and the subtasks too loose. As a result the quality of solutions suffered.

5.6 Conclusion

An execution framework was presented that allows for decomposition of multi-agent problems which require coordination and synchronisation into inter-related single agent problems. This decomposition reduces the size of the plan space and so allows for faster planning. Some of the single-agent problems created by the EF rely on the planning output of some of the other single-agent problems. Such plans must be computed serially. However, problems that share no such link can be solved in parallel. This further increases the speed at which plans are produced. By increasing the speed at which plans are produced less time needs to dedicated to planning and more time can be spent on plan execution. This is especially important if the amount of time elapsed during planning means certain deadlines cannot be met any more, or that an agent no longer has sufficient time to react to a known future external event. When this happens the quality of possible valid plans is reduced, and the amount of available reward decreases.

It was shown that the execution framework was able to find good solutions to problems more quickly than a centralised planner. Indeed, had the centralised planner would not have been able to provide solutions when the time budget was limited as it spent longer planning than the entire time budget. The execution framework was able to handle significantly more complicated domains than the centralised planner. This was both true

of problems with larger states (increased grid height leads to larger states) and problems with more agents.

The execution framework also dealt with problems in which different parts of the goal had different (non-uniform) deadlines. The execution framework was to manage the cooperation and synchronisation necessary in such domains. However, the execution framework may not have performed as well as it could have due to overly loose coordination in the oversubscribed problems.

5.6.1 Further work

One area in which the execution framework could be expanded would be the how agents calculate costs for tasks during allocation. When calculating the cost the planner starts with an empty plan. The planner must attempt to find a plan that achieves all the tasks currently assigned to the agent and the task currently being allocated. There is likely much similarity in the state space of the previous set of tasks allocated to the agent, and the state space of that same set of tasks plus one more task. A planner which attempts to improve and already existing plan could help to reduce unnecessary repetition of work.

As the number of goals in the problem increases so too does the required allocation time. With too small an allocation time, agents may fail to be able bid for goals allocated later in the allocation process. This being due to the planner being unable to compute a plan that achieves all the agents currently allocated goals and the goal currently being allocated. In large problems total allocation time came to dominate planning time. This balance of the time budget between allocation and planning may not be optimal. Further work is required to investigate if there exists a better balance between time spent allocating tasks and time spent planning.

It may be that the execution framework is over-allocating tasks. This is because if an agent is allocated a task then it immediately issues subtasks that fulfil the necessary preconditions. When the agent bids on a second task it attempts to solve both its currently assigned task and the task that is up for auction (this was to assist with load balancing how

tasks were allocated to agents). Whilst the planner is required to compute relaxed plans that achieve both tasks, it is not constrained in how it achieves either task. This means that the resulting relaxed plan might achieve the currently assigned task in a different way. This further means that it might produce different preconditions for solving the same task. That is, more subtasks are produced than necessary and will exacerbate the problem of both too much time being spent on allocating tasks and the problem of agents coordinating which subtasks to achieve. A better way to approach task allocation is to allocate primary tasks first. Only then, once the ancillary subtasks that are required to achieve the primary tasks have been finalised, should the subtasks be allocated.

CHAPTER 6

COMPARISON OF EXECUTION FRAMEWORKS

6.1 Introduction

This chapter will compare the execution frameworks (EFs) discussed in Chapter 4 and Chapter 5. These two different types of execution framework take different approaches to optimising reward gained when sharing a time budget between planning and execution. This chapter will show that these different approaches result in different strengths and weaknesses, but also why they cannot be easily combined.

In Chapter 4 (Acting Whilst Planning During Continuous Execution – AWP) an execution framework (AWP EF) was discussed that increases reward by acting whilst planning – according to one of the discussed heuristics (see Section 4.3). This means goals can be completed faster and, as a result more goals can be completed in a given time budget. To be able to act whilst planning, the planner must be aware of the resultant state changes of acting whilst planning. This is done by predicting the future state that result from the actions executed whilst planning. It needs a fixed planning time, because without a fixed planning time, the initial state cannot be predicted reliably. This execution framework is also centralised and so is free to schedule cooperation between agents in the multi-agent problems in any way possible. What will be shown is that whilst this execution framework can produce plans with shorter makespans than the CS EF (discussed in the next paragraph) it also does not scale well. This is due to the centralised approach that

is taken.

In comparison, the execution framework discussed in Chapter 5 (Cooperation and Synchronisation in Multi-Agent Environments – CS, CS EF) attempts to increase reward gained by decentralising planning. This works by splitting the goal up into subgoals that can be allocated to individual agents. This allows agents to work on a smaller state space (by only considering their own actions and allocated goals). What is distinctive about this approach is that agents are not fully isolated during planning. During the goal allocation sub-phase of planning, agents can request other agents to do certain actions or achieve a given state or subgoal that will enable them to complete their own goals. Once the allocation sub-phase is complete, each agent starts planning for the goals it was allocated, and without further interaction with other agents. The downside of this approach is that the execution framework is not complete (will not find all plans) and not optimal (may not find the best plan). This will be discussed later in Section 6.5 and specific examples given. However, in comparison to the AWP EF, what it can do is produce plans for larger problems where cooperation is required, and can also find solutions much more quickly than the AWP execution framework. This allows the execution framework to attempt larger problems given limited computational resources.

The reason that the centralised and decentralised approaches cannot be combined is that they have different requirements when it comes to planning. As previously mentioned, the AWP EF requires a fixed planning time, whereas the CS EF requires a variable planning time. This is due to the fact that agents are able to create new subgoals if they need assistance completing one of their own subgoals. This means there will be a variable number of subgoals to be allocated that are not known ahead of time. Further, agents that have created such subgoals cannot begin to plan until they know exactly how and when their subgoals will be achieved. This cannot be known before the agent attempting to achieve the subgoal has completed planning itself.

6.2 Domains

The execution frameworks will be compared for performance using two different domains, the Janitor domain from Chapter 4 and a modified version of the Trucks domain from IPC-2006 [11] (chosen for its similarity to the Robocup Rescue domain of Chapter 5). The Trucks domain was modified to require agents to cooperate to achieve goals. As such, both domains require agents to cooperate to achieve all the main goals of the domain. The two main differences are: whether the domain has homogeneous or heterogeneous agents, and the type of cooperation involved. The Janitor domain has homogeneous agents and the execution framework must tightly synchronise when two agents will perform an action together at the same time. By comparison, the trucks domain has heterogeneous agents, and the goals in this domain need less tight synchronisation. The difference being that agents in the Trucks domain agents must ensure some of their actions have an overall order (ie. Action A finishes before Action B starts), so that state changes be temporally consistent. However, with the Janitor domain, agents must schedule some actions to occur at the same time as another agent (ie. the extra dirty action [see Section 4.1.2]), which the results will show is a more complex optimisation problem (when taking into account multiple goals). The reason that cooperation and synchronisation is required in the Trucks domain is that, as heterogeneous agents, the agents are not capable of all actions. Further, problems are posed such that the set of actions available to an individual agent are not enough to achieve their goals on their own. Cooperation is achieved by finding actions that are required to achieve goal, but that the agent is unable to achieve the preconditions of these actions. These states are turned into new subgoals that are allocated to other agents.

The Janitor domain experiments, with its tight synchronisation, will demonstrate that planning over all agents using the centralised EF is important to producing better plans for domains with tight synchronisation. That is, the centralised execution framework has absolute freedom to reorganise the schedules of the agents it is planning over. Whereas, in a decentralised execution framework, synchronisation timings cannot be changed in an

isolated planning phase of a single agent without requiring any agents affected by the synchronisation timing change to replan. By definition a decentralised planner only reasons over a subset of agents or actions. As such a decentralised execution framework is unable to reason about globally optimal synchronisation times. Rather, it has to optimise its part of synchronisation locally. This can result in a suboptimal global plan as the locally optimal timing for a synchronised action for one agent may lead run-on effects in other agents – wasting time of other agents and being globally suboptimal. In comparison, the loose coupling of the Trucks domain will be shown to be better suited to the decentralised execution framework. By only planning over the actions of one agent, a decentralised execution framework (distributed over multiple processing cores) is able to search over a larger state space.

6.2.1 Trucks Domain

To help compare the two different execution frameworks being discussed in this chapter, we used the transport domain from IPC-2006 [11]. In this domain the problem is to use a fleet of trucks to deliver a set of packages from their pick-up points to their destinations. The actions in this domain are 'move', 'load', and 'unload'. Trucks can load multiple packages at the same time, but must unload packages in a first-in-last-out (FILO or stack) order. This enables an agent to work towards achieving multiple goals at the same time (moving two loaded packages towards their destination rather than one at a time). On its own, this domain does not require cooperation between agents. However, this domain was used as it is the IPC domain that is simplest to turn into a multi-agent problem requiring cooperation. Each truck can be considered an agent that can be given a set of packages to deliver. A modification was made such that are two types of transport agent (as opposed to just one). Each type of agent is able to traverse a different edge type. As such, a given package might require multiple agents to cooperate in order to deliver the package from its initial pick-up location to its destination. This could be a way of modelling separate truck and ship networks where each agent can only traverse

the land or sea, or as multi-national delivery network where each truck must stay within its national network.

A new set of problems were created for this domain since it had a new agent type and edge type, and was, therefore, incompatible with the problems supplied with the original domains. These problems were created in a square grid that was horizontally separated into two distinct "networks". The two networks were sparsely connected with each other – only being connected at the edge of the grid (ie. on the east-most and west-most edges). The variables for each problem were grid height, number of agents, and number of packages.

Cooperation

Cooperation in this domain was similar to the Robocup Rescue domain in Chapter 5, with its heterogeneous agents. Each agent type has actions that are unique to its agent type. In the Trucks domain, this is manifested as move actions that enable the agent to traverse certain edge types in the graph. To establish if cooperation is required, an agent during the bidding process must see if it can move the package to the destination on its own. If it cannot, it enters a heuristic planning phase where it places an imaginary agent of the opposite type at the starting location of the package. This imaginary agent is allowed to load the package and unload the package that is being bid for precisely once. This is so that only one new subgoal is created by the bid. That is, an agent bids that it can deliver a package, subject to another agent delivering the package to a transition point between the two networks first.

Any heuristic plan that does not involve the bidding agent is rejected automatically. That is, such a bid only involves the imaginary agent, and so represents an agent saying it can ensure delivery of package if only another agent does all the work. As such, bids for such packages should be bid for directly by agents of the corresponding type.

In contrast to the Robocup Rescue domain there is no strict hierarchy about which type of agent can assist the other type of agent. Instead, the direction of the cooperation

between the agents is dependent on which network the package starts in and which it is to be delivered to. The following example will demonstrate the mutual cooperation between the agent types. Given two nodes n^x and n^y with transition point n^t, and two agent types a^x and a^y, if a package starts at n^x and is to be delivered to n^y then a^y requires that a^x deliver the package to n^t if it is to be able to deliver the package to n^y. Had the package started at n^y then the reverse would have been true. This creates problems with the ordering of which agents plan first after bidding. That is an agent achieving the subgoal of another agent must plan how it will achieve that subgoal before the second agent can start making plans dependent on when the package arrives at its subgoal node. If the two agents mutually depend on each other for different goals then they planning is deadlocked.

To prevent deadlock in the planning process, the bidding process is modified to keep track of the bidding state of each agent. The bidding state values can be "independent", "won bid with requirements" or "won assist bid". The bid state "won bid with requirements" means the agent has won a bid for which it requires assistance from another agent to deliver the package to an agreed upon intermediary point (a subgoal node). Such agents need to know when such packages will arrive at the subgoal node before it can plan. An agent with such a bid state is not allowed to bid on subgoals – this prevents it from getting into deadlock. In comparison "won assist bid" represents that the agent has won such an assist subgoal. This agent must finish planning before the agent it is assisting can plan. As result, agents in such a bid state are not allowed to create bids that creates a new subgoal. Finally, "independent" is the starting bid state of all agents. It represents agents that neither require assistance nor are providing assistance. It is entirely independent of other agents, and can bid for subgoals and create bids that have subgoals – though doing so will change its bidding state. By using this bidding state, the bidding process is able to create dynamic teams of agents that are better able to deal with different problems. That is, given all packages were located in network n^x and needed to be delivered to network n^y, a fixed allocation of agents would have some agents unable to bid for anything. For

instance, agents in n^y assigned to assist agents in n^x with moving packages from their start node in n^y to their goal node in n^x would have no work to do and sit idle for the duration of the problem. However, a dynamic allocation of agents to different assistance types allows for all agents to be productive.

6.2.2 Janitor Domain

By default the decentralised execution framework cannot run on the janitor domain. This is because the domain as specified requires the planner to be able to reason over all agents. That is, the "extra dirty" action requires two agents to work simultaneously on the action. To address this issue for the decentralised execution framework the "extra-dirty" action was separated into two distinct parts. A main clean action that requires a co-operation predicate over its duration, and an assisting that provides the co-operation predicate for its duration. The PDDL form of these actions can been seen in Figure 6.1. This way of separating out the action of cleaning "extra-dirty" rooms allows planning to be decentralised. First, bidding takes place where the cooperation precondition (the *cleaning-assist* predicate) is removed. The agent that is awarded the goal to clean the "extra-dirty" room then adds a new goal to achieve the *cleaning-assist* predicate. The assisting agent that is awarded the cleaning assist goal then plans to achieve all its goals, including the cleaning-assist one. Once the assisted agent has a plan, the timing of the *cleaning-assist* action is extracted and this information included in the initial state information for the original agent planning problem. In PDDL this takes the form of a Timed Initial Literal (TIL) that becomes true at the given moment and lasts for the duration of the cleaning action, when it becomes false again. Thus, the only way for the original agent to complete the goal of cleaning the "extra-dirty" room is to be in the same place at the same time as the assisting agent, at which point they can carry out the action.

```
(:durative-action extra-clean
        :parameters (?a - agent ?rm - room)
        :duration (= ?duration (dirtiness ?rm))
        :condition (and
                (at start (extra-dirty ?rm))
                (at start (available ?a))
                (over all (cleaning-assist ?rm))
                (over all (at ?a ?rm))
        )
        :effect (and
                (at start (not (extra-dirty ?rm)))
                (at end (cleaned ?rm))
                (at start (not (available ?a)))
                (at end (available ?a))
        )
)

(:durative-action extra-clean-assist
        :parameters (?a - agent ?rm - room)
        :duration (= ?duration (+ 0.001 (dirtiness ?rm)))
        :condition (and
                (at start (extra-dirty ?rm))
                (at start (available ?a))
                (over all (at ?a ?rm))
        )
        :effect (and
                (at start (cleaning-assist ?rm))
                (at start (cleaning-assisted ?rm))
                (at end (not (cleaning-assist ?rm)))
                (at start (not (available ?a)))
                (at end (available ?a))
        )
)
```

Figure 6.1: Action description for a decentralised and synchronised domain

6.2.3 Decentralised Execution Frameworks and Synchronisation

The task allocation algorithm (TAA) described in Section 5.3.4 does not work well with domains that require direct synchronisation. This is because the algorithm only considers deadlines – when a task must be completed by. In fact, it would be considered beneficial if a task is completed earlier than required, as this is latest at any plan that depends on the subgoal can be achieved. In direct contrast, the Janitor domain requires that a cooperating agent execute an action at a specific time – not before and not after. The agent issuing the subgoal cannot provide the specific time at which the action to achieve the subgoal should occur as it does not know what is possible for the other agents. It can be assumed that the agent that wins the bid is the one that can start the required action at the earliest opportunity. As such, the schedule proposed by the agent that achieves the subgoal will be acceptable to the issuer of the subgoal. This means that subgoals must are issued without any timing information whatsoever. Any subgoals produced during task allocation must be added to the list of goals to auction in a last-in-first-out manner. This is to make sure there is the highest chance of an agent being available to provide assistance. The original implementation uses deadlines to decide which goal to auction next. As the subgoals have no deadlines they will always be auctioned last, when it might be that all agents already have a won a large set of goals, and so the time allocated to task allocation is insufficient to come up with a plan for these subgoals.

Allocation with homogeneous agents also provides a new problem. With heterogeneous agents the distinction between assisted and assister is clear, and it is also clear that some agents must plan after others as they rely on the planning output of said agents. With homogeneous agents this distinction is not so clear, but some agents still require the planning output of other agents to be able to plan. As such agents should be differentiated based on whether they have issued a cooperation subgoal or have been allocated such a goal. Any agent that has issued a subgoal must wait for the agents that are tasked with achieving the subgoal to finish planning, before the assisted agent itself begins planning. However, this raises a new problem: what if two agents are mutually dependent on each

other to provide cooperation to each other on separate tasks. That is, agent A is assisting agent B with goal X and agent B is assisting agent A with goal Y. These agents will be in a deadlock, each agent waiting for the other agent to finish planning before they begin planning. To prevent this problem from occurring in the first place, agents are not allowed to issue subgoals if they have already been allocated a subgoal, nor is an agent allowed to issue a subgoal if it has already been allocated a subgoal.

6.3 Method

The experiments were run on Amazon Web Service EC2 t2.micro instances. Each instance has a CPU with a clock speed of 2.5 GHz and 1 GiB of RAM.

In chapter Chapter 4 10 seconds was chosen as the planning time for the Janitor domain. However, the AWP EF (centralised) with a 5x5 grid a 10 second planning time failed to find any plans for such problems. Other planning times that were tried were 30, 60 and 120 seconds. Above this, the machine would run out of RAM before planning finished – meaning performance would deteriorate. Of those, the 120 second planning time found solutions to 5x5 grid problems most frequently and so was chosen for grid sizes 5x5 and above. The PR-New heuristic is the best heuristic for the AWP EF (see Section 4.3). However, the frequency with which planning failed with planning times less than 120 seconds means that it frequently fails to provide actions to execute whilst planning. Instead, the PR-Reuse heuristic was used because it does not require planning to provide actions to execute whilst planning. This heuristic attempts to reuse as much of possible of the pre-existing plan when replanning. This heuristic is safe for both the Janitor domain and Trucks domain as there are no negative irreversible state changes in either. The Trucks domain problems were found to be harder than the Janitor domain and so the maximum 120 second planning time was also used.

For the CS EF a planning time of 5 seconds and a allocation planning time of 1 second were chosen. The allocation time was chosen as it was the same as used for the Robocup

Variable	Values
Width of Graph Layout	3, 4, 5
Number of Nodes	9, 16, 25
No. of Packages	4, 8, 12
No. of Trucks	3, 4, 5
No. of Boats	3, 4, 5

Table 6.1: Possible values of variables, with co-varying variables grouped.

Rescue domain in Chapter 5. Five seconds was chosen for the planning time as it is half of what was used for the Janitor domain used in Chapter 4. That is, 5 seconds for each assisting agent and 5 seconds for each assisted agent. For the Trucks domain the same allocation times and planning time were chosen. The CS EF may have performed better with an increase to these values, but as the planning time for the AWP EF on the Trucks domain could not be increased further it was decided to keep planning times equal across both domains.

6.3.1 Trucks Domain Variables

The different grid heights used were: 3, 4 and 5. The number of agents used were 3, 4 and 5. The number of packages was tied to total nodes in the graph (and therefore grid height) at half the number of nodes in the graph.

6.4 Results

The results section shows the performance of the AWP and CS execution frameworks on the two different domains. The performance metrics are completion time and success rate. Success rate is defined as the proportion of problems that the executor was able to complete all goals for. A run is stopped once all goals are achieved or when, after task allocation and planning, there exists no agent with a plan that will accomplish at least one goal or subgoal. If a run is stopped for the latter reason then it is deemed to have failed. On larger problems the execution framework might be able to achieve a few goals

that are relatively simple to achieve, but is unable to find plans to achieve goals that require more complex (longer) plans. Completion time is the average time from start to completion of all goals, inclusive of any time spent planning. A good EF will have lower completion times, proportional to problem complexity, and high success rates.

6.4.1 Janitor

For the Janitor domain, it is shown that the AWP EF has lower completion times than the CS EF. The lower completion times are consistent across all problems sizes for which the AWP EF had a non-zero success rate. However, the completion times are only marginally lower. From even from the start, planning for the AWP EF takes up a significant proportion of the total time to achieve all goals (see Figure 6.3). Whereas, for the CS EF this only becomes problematic for much larger problems. That is, the planner only finds a partial solution on the first iteration of planning, as opposed to a full solution that achieves all goals. The agents execute as much of this partial plan as they can, at which point the planner is asked to find a new plan for the remaining goals, and so on until all goals are completed. By the time we reach a grid height of 6, the AWP EF has only managed to successfully complete 1 problem out of 30. For this problem it was planning for 840 seconds of the 1656 seconds it took to achieve all the goals. This shows that the ability of the AWP EF to find good plans drops off rapidly as problem complexity increases.

The CS EF has significantly higher success rates than the AWP EF. In fact, the AWP EF is only able to produce solutions for problems with a grid height of 6 or less. Whereas the CS EF still produces solutions for problems with a grid height of 8 or less. It should be noted that the complexity of the problem increases considerably as grid height increases. That is, the size of the network to be traversed, the number of goals, the number of goals requiring cooperation are all related to the square of the grid height. This is because the Janitor problem uses a square layout of nodes, and each node aside from the starting node must be cleaned. Further, the number of goals requiring cooperation is linearly

proportional to the number of goals. Thus, larger grids require more cooperation and synchronisation between agents. As there are multiple agents, the state space that needs to be considered by the AWS EF increases even more quickly with grid height. For example, a problem with grid height 4 and 3 agents has $(4^2)^3 = 4096$ different spatial configurations of agents. Whereas, a problem with a grid height of 5 and 3 agents has $(5^2)^3 = 15625$ different spatial configurations of agents – a nearly 4-fold increase.

6.4.2 Trucks

With the Trucks domain the AWP EF and the CS EF were unable to find solutions to problems with grid sizes as large as the Janitor domain. This shows that the Trucks domain is significantly more complex than the Janitor domain. This is due to the complexity required to achieve a goal. In the Janitor domain an agent need only move to a location and then perform an action to achieve a goal. By comparison, an agent in the Trucks domain must first move to a package start location, load it, move to the package destination, and then unload it. This is further complicated by the fact that agents are able to reason about loading and moving multiple packages at the same time.

In contrast to the Janitor domain, the CS EF resulted in better completion times than the AWF EF. This is in addition to the CS EF maintaining a higher success rate than the AWF EF – having a higher success rate for the same problems, but also being able to solve more complex problems with higher grid sizes. By agreeing on synchronisation points between heterogeneous agents, planning can be simplified, and the size of the state space the planner is required to reason about significantly reduced.

The total planning time of both the AWP EF and CS EF is low compared the completion time. Especially in comparison to the Janitor domain experiments. The reasons for which are discussed in Section 6.5.

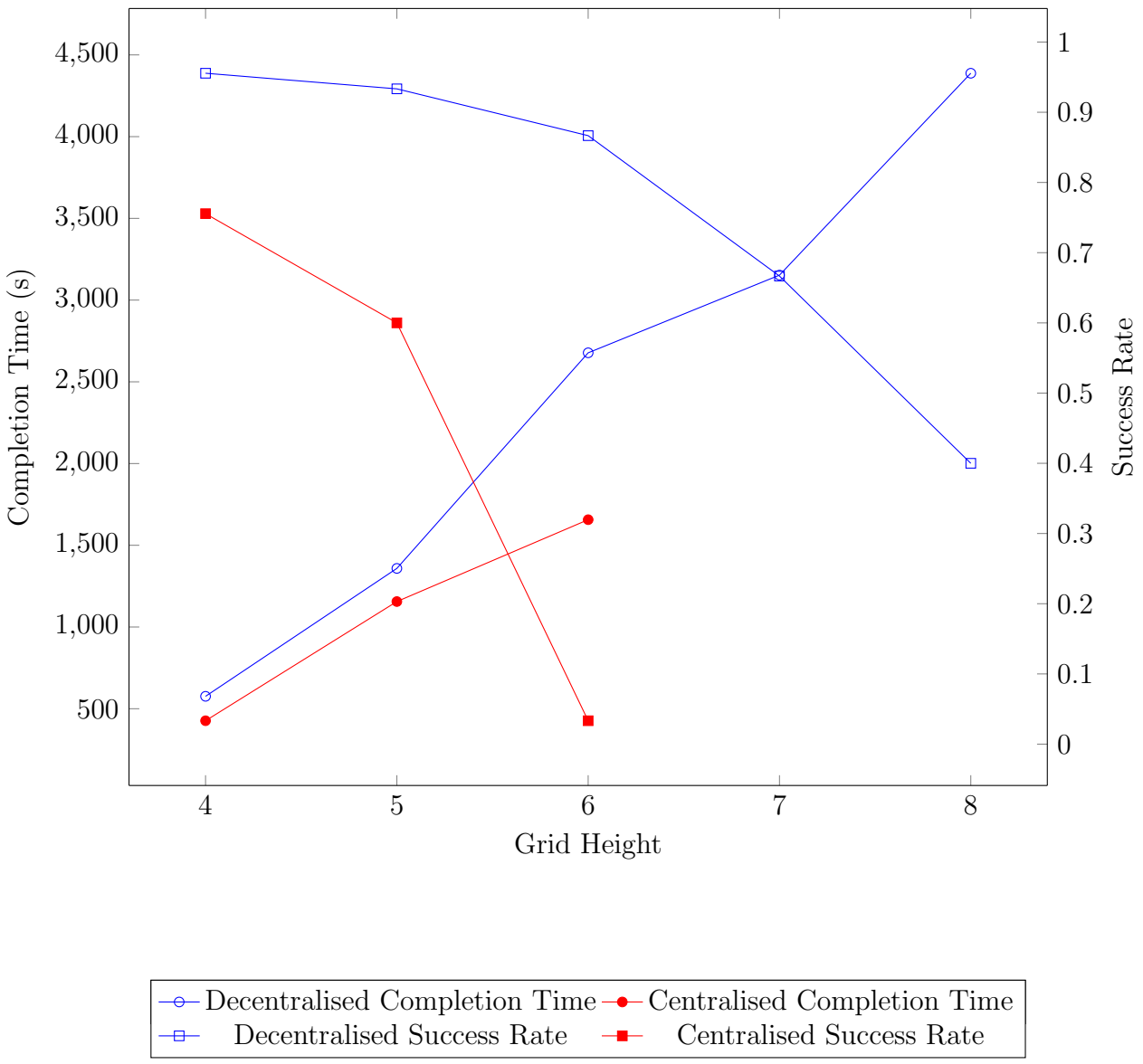

Figure 6.2: Completion Time and Success Rate of centralised and decentralised EFs on the Janitor domain against Grid Height

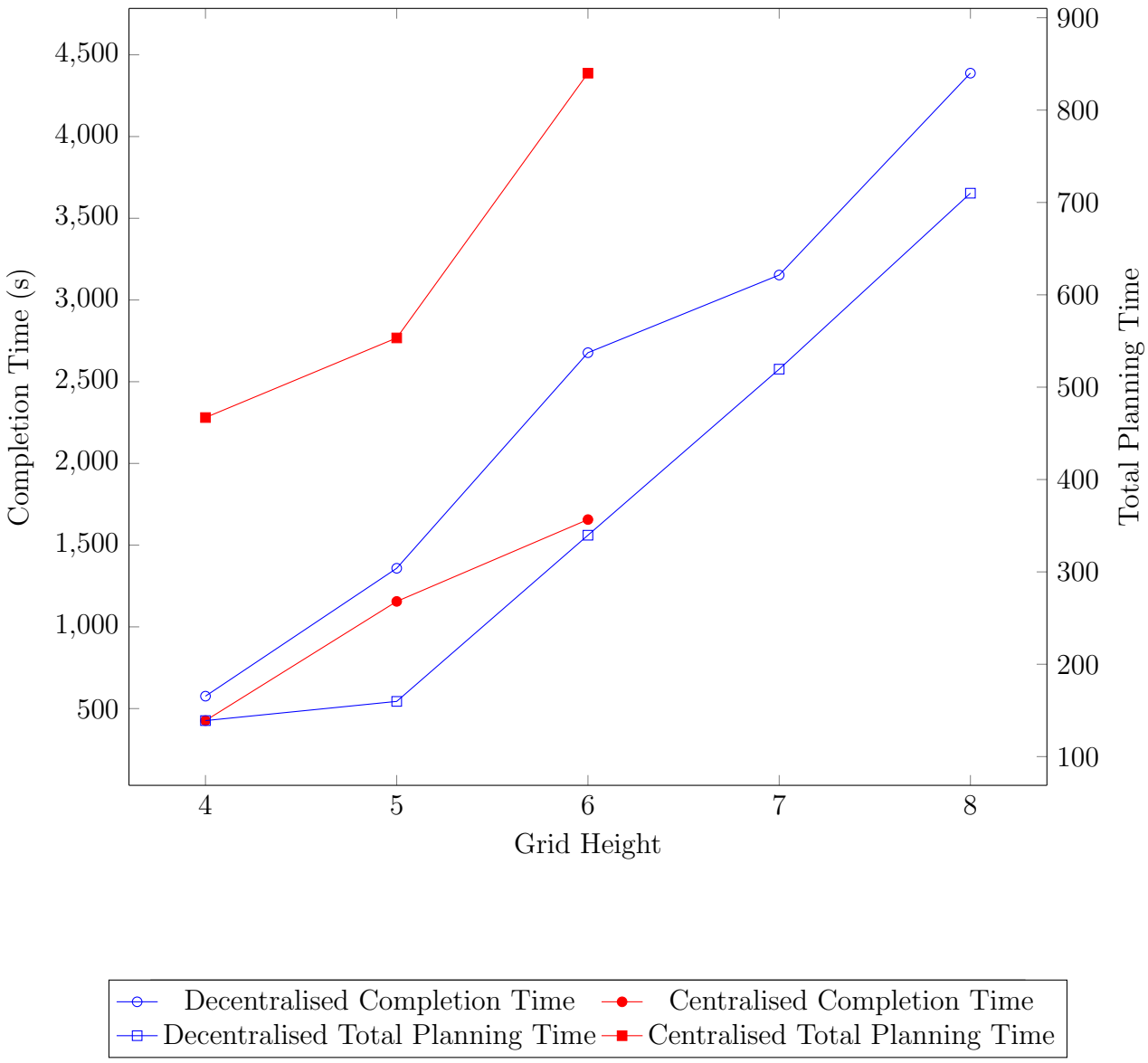

Figure 6.3: Total Planning Time and Completion Time of centralised and decentralised EFs on the Janitor domain against Grid Height

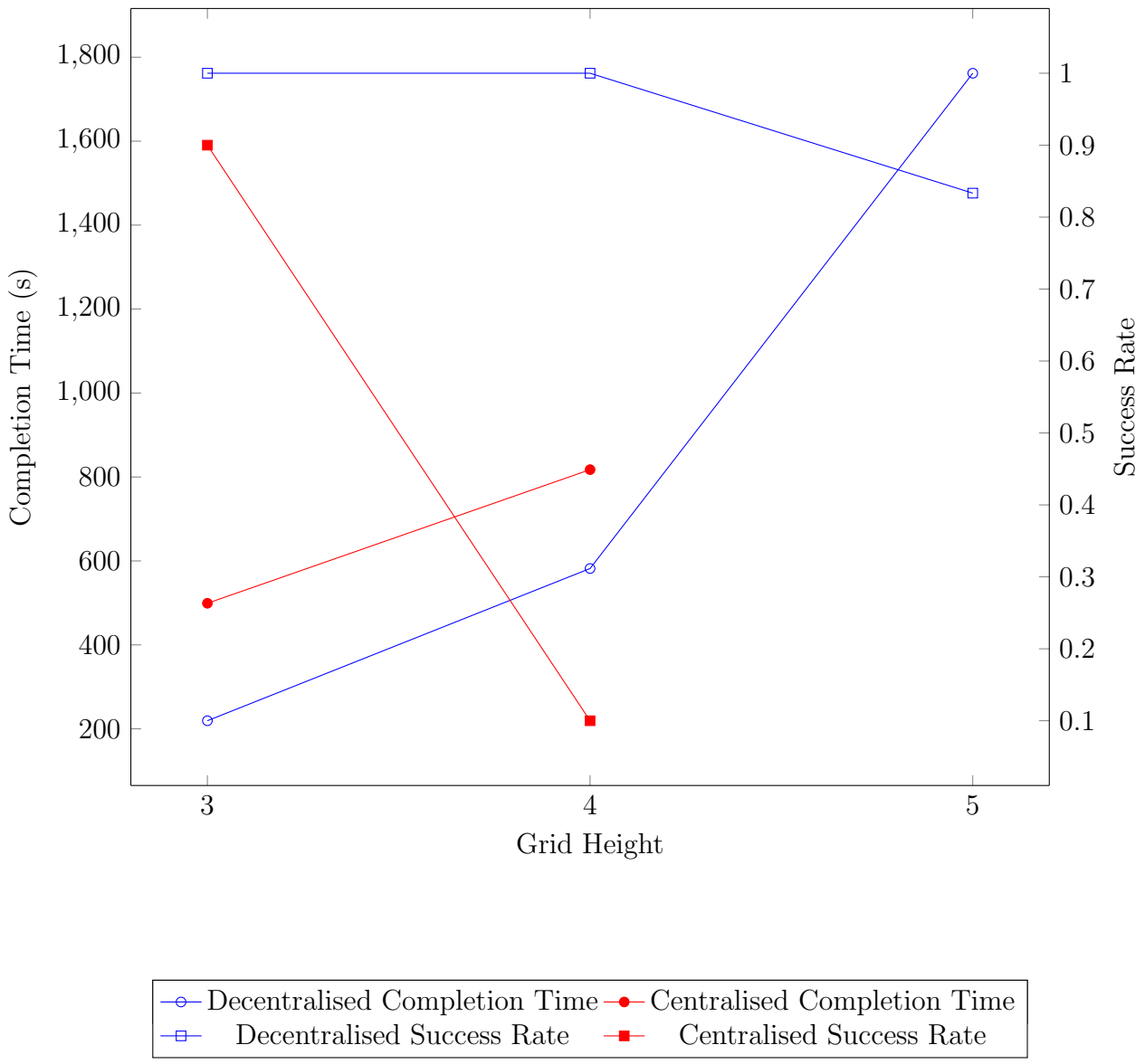

Figure 6.4: Completion Time and Success Rate of centralised and decentralised EFs on the Trucks domain against Grid Height

6.5 Analysis

An investigation into why the planning times were significantly higher in for Janitor domain CS EF (Cooperation and Synchronisation EF) shows several reasons why. In execution logs, it was shown that "extra-dirty" rooms were usually the last to be cleaned, but that during allocation they were always allocated, and the cooperation subgoal allocated. That is, when it came to individual agents trying to create plans for their own goals (including the cleaning the "extra-dirty" rooms) the cooperating agent is responsible for picking the precise time. It will choose a time that fits best achieving its other goals. This leads to local optima where individual agents are maximising their reward at the expense of maximising global reward. The agent requesting cooperation could be required to wait for an extended period of time before the cooperating agent arrives. Frequently, however, agents would instead make plans to go off and clean the rooms they can clean by their self. This is reasonable, as it would lead to higher reward. That is, an agent cleaning two rooms by itself gets more reward than waiting to clean one "extra-dirty" room. This is exacerbated by the fact that the total reward to clean an "extra-dirty" room is equal to cleaning a normal room, and so when this reward is shared between the two agents it is valued less than the cleaning of normal rooms. Given longer planning time agents would sometimes be able to find a plan that included cleaning the "extra-dirty" room. However, such plans had many more actions due to the agent having to move away from the "extra-dirty" room only to return again later. All of this together shows that decentralised planning does not cope well with the tightly coupled cooperation required by the Janitor domain.

The CS EF uses decentralised planning and as such lacks a full awareness of the scheduling constraints of the actions of an agent. Agents are allocated goals based on

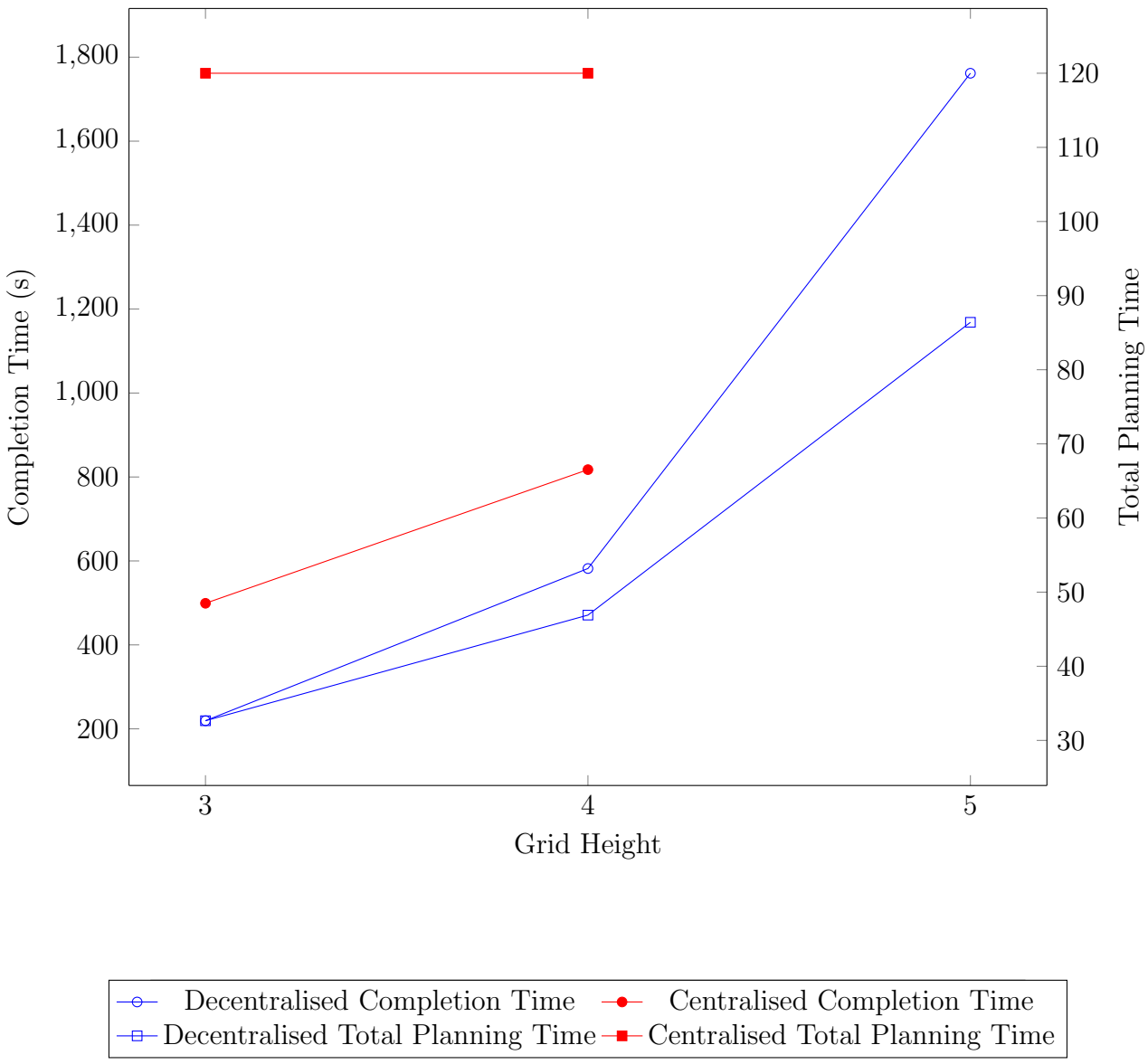

Figure 6.5: Total Planning Time and Completion Time of centralised and decentralised EFs on the Trucks domain against Grid Height

the earliest time which they can achieve that goal (and all other goals that have been previously allocated). However, it may be that no other agent will be able to cooperate with the agent for a long time because it is far away and must move towards the first agent before it can assist. A centralised planner would be able to detect this and plan for a set of agents to cooperate on a goal at a time that is optimal both locally for both agents, but also globally. The decentralised planner is only able to optimise locally for each agent, and such local optimizations might be globally suboptimal, or even result in not being able to produce a valid plan. That is, in the decentralised CS EF, given three agents a_r, a_{p1}, a_{p1}. Where a_r is the agent requesting cooperation, and a_{p1}, a_{p1} are agents providing cooperation. a_r is allocated two goals that require cooperation. Agents a_{p1} and a_{p1} are both allocated one of these cooperation subgoals each. As no timing information has been provided for these subgoals, the agents providing assistance are free to schedule to achieve these goals as it suits them. In this instance it, as they are the agents' only goals, they will achieve them as soon as possible. However, these goals might be distant from each other, that means if both a_{p1} and a_{p1} provide the assistance at similar times then a_r may not be able to make use of both the agents providing assistance. In short, a_r cannot be in two places at the same time. This was seen to happen at least once in the Janitor problems, where an agent provides assistance only for no agent to make use of it.

The Trucks domain had a much higher failure rate than the Janitor domain. Neither EF was able to cope with as large a grid size for the Trucks domain as it could for the Janitor domain. This is because in the Janitor domain the state of each agent is very simple – it is only its location. However, in the Trucks domain each agent also has a capacity to hold multiple packages, and so for an agent with two packages ($p0$ and $p1$), it can have 5 states before location is even factored in (empty, $p0$ loaded only, $p1$ loaded only, $p0$ loaded first and $p1$ loaded second, and $p1$ loaded first and $p0$ loaded second). A more minor cause is that the Trucks domain has slightly more packages requiring cooperation for a given grid size. In the Janitor domain, a problem with a grid size of N had $\frac{N^2}{5}$ goals which require cooperation. Whereas, the Trucks domain has $\frac{N^2}{2}$ goals which require

cooperation. Further, the allocation for such goals requires multi-agent planning in a way the Janitor domain does not. That is, during allocation in the Trucks domain, an agent must compute the intermediary location that the agent requesting cooperation is able to pick up the package and that the agent providing cooperation is able to drop off the package at. To do this, the allocation planner must be able to reason about a second agent with different movement restrictions to the primary agent. If not all agents can reach all locations then this problem is non-trivial. Even when all agents can reach all locations, optimising over which intermediate location to exchange the package is non-trivial too.

6.6 Discussion

The difficultly of computing intermediate package exchange locations proved to be more difficult than anticipated, and not enough time was given to the allocation algorithm. Further, very little domain information was provided to the allocation algorithm in how to achieve this. That is, the allocation algorithm was given one additional agent starting at the package location. It was able to load and unload the package exactly once. With that the allocation algorithm was asked to find a plan that could deliver the package (as well as any other packages it had to deliver). If it could, then the location that the packaged was unloaded at was used when creating the new subgoal of delivering the package to an intermediary package exchange location.

Problems with the Janitor domain scheduling arose when trying to achieve an extra-dirty clean goal. This meant the CS EF was not able to complete all small problems initially. This was detected by the simulator finding that some agents were attempting an action from an invalid state. That is, cooperation between two agents was achieved by way of two actions that occur at the same time, but with one having an effect that is a precondition to the other. As a serial processor the simulator had to process the start of one action before the other. It just so happened that this was done in the wrong order and thus the simulator believed that one agent was attempting a cooperative action

without having actually been provided with cooperation. This was corrected by adding a second metric by which the simulator sorted how to process actions. That is, actions were primarily sorted by time and then and by action type. This would ensure that assisting actions would always start before assisted actions that started at the same instant.

A specific limitation of the implementation of the CS EF is that it cannot create plans which require cooperation from 3 or more agents. That is, given a Trucks domain problem, the CS EF cannot create a plan that would have package starting in one network, be transported across a second network before finally being delivered to its goal location in a third network. One example of this is delivering a package from the middle of one island to the middle of another island. The package must first be delivered to port by truck, then transported by sea to a second port, and then finally transported by truck to its final destination. This is a because the CS EF algorithm is overfitted to the specifics of the Janitor and Trucks domain problems. There is no reason the allocation algorithm cannot be expanded to reason about 3 or more agents. However, increasing the number of agents the allocation algorithm reasons about will increase its computational complexity. As noted above, this is already an issue with just two agents. The allocation algorithm is required to be fast as it is invoked separately for each agent for each goal. If it is not fast it can quickly come to dominate total planning time. Finding simple plans quickly is also useful to the AWP EF. This is because the PR-New heuristic gives the best results. For the PR-New AWP EF to be effective it must be able to find simple, if suboptimal plans, in a fraction of the time dedicated to planning overall.

6.7 Conclusion

In this chapter the execution frameworks presented in Chapter 4 (Acting Whilst Planning During Continuous Execution) and Chapter 5 (Cooperation and Synchronisation in Multi-Agent Environments) were compared against each other on two separate domains. One domain (Janitor) from Chapter 4 was reused and adapted. The second domain was

the Trucks domain from IPC-2006. A small modification was made such that the domain is relevant to the problem that Chapter 5 addresses, namely managing multi-agent cooperation. The two domains represent tight synchronisation and loose synchronisation respectively. Also discussed was why these two execution frameworks cannot be combined. This is because they each have different requirements to how planning time is set. The AWP EF requires that the amount of time to be spent planning be known before planning begins. Whereas, the CS EF is not capable of predetermining how much time will be spent planning. This is because it does not know how many subgoals will be created during the planning process and therefore, how much time will be needed for allocation of all goals.

This chapter showed that the CS EF scales better than the AWP EF due to its ability to synthesise smaller, inter-related sub-problems from whole problem and solve these separately. However, this comes at a cost, this being that the CS EF is not complete and is unable to find all solutions to a given problem. Instead, when planning time is limited, it finds solutions with more reward than the more centralised approach of the AWP EF. In contrast, the AWP EF is complete, and will find a solution given enough time and memory. As such, given smaller problems and enough planning time it will find better, shorter solutions than the CS EF.

In Section 6.6 the limitations of both EFs were discussed. The AWP EF increases reward by scheduling actions during planning. However, it requires certain properties of the problem and/or domain to be effective. These properties being specific to the heuristic being used. For PR-New it requires that an initial plan can be found quickly. In comparison, the CS EF has problems with tight synchronisation. The requirement to have a specific action at a specific time is not well suited to the planning algorithm that was used.

CHAPTER 7

DISCUSSION

This chapter discusses the strengths and weaknesses of the approaches of described in Chapters 3 to 6, and also the merits of each approach in comparison to the others.

7.1 Optimising Planning Durations in Continual Domains

The loss limiting meta-management system (MMS) described in Chapter 3 addressed the problem of how long should an agent plan for. The loss limiting MMS is a fast and simple way to optimise how much time is spent on planning. It requires only a reward function to be able to compute the amount of reward available from executing the plan at the current time, and a loss tolerance to compare current execution reward against the best execution reward it knows it could have achieved. As it is fast, the cost of monitoring is low and will not lead to a degradation of planning performance. However, because the MMS is loss limiting it can never be optimal. By definition the best reward it can achieve is the maximum amount less the loss tolerance. Despite this limitation, the loss limiting MMS was still better than using fixed planning times that are unable to adapt to the complexity of a specific problem. Its simplicity also means it is easier and faster to implement than the predictive MMS described in Chapter 3 and the MMS described by Hansen and Zilberstein (2001) [30].

Due to the loss-limiting nature of the MMS it requires domains in which new plans can be found in relatively quick succession. The longer it takes between generation of new plans the higher the loss tolerance needs to be. In domains where there is a relatively long time between generation of new plans, but that each new plan has a significantly higher reward the MMS will terminate planning early as too much of the reward from the initial plan has been lost. This could lead to situations where the MMS executes plans of a lower quality than it could have otherwise. However, increasing loss tolerance also has drawbacks. That is, the MMS will only execute a plan after it has lost so much of the reward of the plan. Thus, high loss tolerances lead to the MMS being able to only execute small portions of high quality plans, leading to a lower amount of reward.

The loss limiting MMS works best with oversubscribed domains. That is, in such domains the planner will return plans that span the full length of the time budget, but for which not all goals will be achieved. As time passes, less of the goals that could be achieved by the plan will still be achievable. This gives the MMS a cost of time by which it can analyse whether planning is productive or not.

As the MMS uses a variable planning time, the start time of the plan will not be known in advance. As such, the plan cannot make any assumptions about when an action will happen. This means the MMS is not suited to domains in which scheduling is an important requirement. That is, given a domain which has external events that happen at known times in advance, the planner will have trouble optimising plans before the occurrence of this event. For instance, in the UAV domain, if it is known in advance that some target locations will cease to be visible after certain times then it would make sense to visit these locations first. However, if a plan is optimised to visit as many of these locations as possible, then delaying execution of the plan might a big impact on the quality of the plan. It could mean that none of the locations with visit-deadlines are visited on time, with the agent arriving just too a little too late at each location.

7.2 Acting Whilst Planning During Continuous Execution

Chapter 4 addressed the question of what an agent should do whilst planning. It described a set of execution frameworks that enable the agent to act whilst planning. This leads to an increase in reward as the agent is able to spend more time executing, and thus reduce the amount of time taken to achieve goals. The strengths of this approach allow an agent to better react to changing circumstances. If a plan is invalidated due to an environmental change, the agent is able to both react immediately in the short term, and also plan longer term in response to new information. This technique also allows the agent to react to new beneficial information without having to consider the cost of replanning. Without using this execution framework an agent would be required to stop executing whilst it replans in response to the new information. If there is no new better plan, or the new plan is only marginally better than the old plan, it may have been better just to continue executing the old plan. However, by acting whilst planning the agent can minimise the cost of replanning and always benefit from new information. New goals can be treated as new information. This enables an agent to react to new goals being added without having to stop executing.

To be able to act whilst planning, the execution framework requires a fixed planning time that is known in advance of planning. It uses this to estimate a future state which it can pass to the planner as its initial state. Without this initial state, the planner is unable to reason about that actions that are executed during planning. If planing was terminated early, or allowed to go on for longer, then the initial state that the planner was given might not match the actual state, and thus the plan that was produced could be invalid. Thus, the AWP EF must used a fixed planning time. Variable planning times can be useful though (as discussed in Section 7.1).

Some of the execution frameworks reuse actions from the previous plan to continue executing. This can cause problems if some states are harder to undo than others. In Chapter 4 this was discussed as irreversible actions. Actions that once performed cannot

be undone (such as driving off the edge of a cliff). By reusing the previous plan, an agent is unable to react to the new information that shows that some actions in the plan are now irreversible, and that the plan is invalid. The PR-New execution framework avoids this problem by finding an initial plan as within a short time, and executing that instead whilst replanning. This means that the agent will never execute an irreversible action whilst using PR-New.

PR-New sacrifices some of its planning time to come up with the initial plan. During this time the agent does not execute any actions. Once it has the initial plan, it must restart planning with the new predicted initial state. However, this means that PR-New spends less time planning on the full problem than the other execution frameworks presented in Chapter 4, and so meaning it may not always come up with as high quality plans as it would have otherwise. However, this weakness was found to be offset by more often having a plan to execute during replanning. Sometimes, however, PR-New is not able to find a plan to execute during planning. This is because the time dedicated to coming up with a plan to execute during planning is insufficient to actually find a plan. In such situations PR-New decays to the standard model of planning and then executing the resulting plan.

7.3 Cooperation and Synchronisation in Multi-Agent Environments

The execution framework presented in Chapter 5 was shown in Chapter 6 (as the CS EF) to be more effective at producing solutions to problems than the AWP EF (a centralised planner). It finds solutions to more complex problems and on problems of similar complexity it is capable of finding solutions of similar quality or better. This is despite the advantages of the AWP EF being able to execute actions during planning. The CS EF fairs better than the AWP EF because of its ability to synthesise smaller, inter-related sub-problems from the whole problem and solve these separately. The CS EF is fast,

effective and scalable.

The first problem that the CS EF has is that it is greedy when it comes to task allocation. If an agent says it can achieve a goal *if* another agent cooperates with it, then it is assigned the goal without first checking that another agent is capable of achieving the required subgoal(s). If no agent is capable of achieving the subgoal then the original goal is left unallocated and is forgotten about. In such a situation it might be better to back track and try allocating the goal to another agent to see if that would work. The execution framework mitigated this problem by preferring to allocate goals to agents that did not require cooperation. However, the initial reason for doing this was that sometimes agents would bid for goals based on plans that required another agent to do all the work. As such, the task allocation algorithm always allocated goals to an agent that did not require cooperation if it could. It is possible that this might not be the global optimal allocation, but it ensures that no goal was unachievable due to a failure in the allocation process.

A second problem with the execution framework is that it is optimistic about scheduling. That is, subgoals contain no information about how they relate to each other. Thus, an agent that was allocated two subgoals may fulfil the subgoals at such a time or in such a way that the agent that issued the subgoals cannot make use of both of them. For example, take two goals (g_0 and g_1) which must be worked on serially (eg. two packages that must be delivered to different location in opposite directions). Each goal has a distinct deadline and minimum makespan of the plan required to achieve them, this implies that there is a minimum starting time for an agent to be able to achieve both goals (s_0 and s_1). To achieve both goals the agent must achieve one of the goals before the minimum start time of the other goal. However, the order in which the goals are achieved is not important. This is a high level concept that is not encoded in the subgoals an agent issues when it requires assistance. Instead, all the agent communicates is the maximum time by which that particular subgoal must be achieved. Any agent allocated these subgoals would not know about the dependency between them. Indeed, without

analysis the issuing agent will not know about this dependency. Even if the agent could know about the dependency between these subgoals, there would be problems if the subgoals were allocated to separate agents. That is, for an agent to know if it has achieved a subgoal by a maximum required time it must know when the other agent will achieve the other subgoal by. This problem could be overcome by assigning both subgoals to the same agent, but it might be the case that only two agents acting separately can achieve both subgoals by the required maximum times.

Problems around scheduling were a problem for the Janitor domain in Chapter 6. Because of the exact start and finish times of the subgoals, agents would sometimes find that the subgoals it issued were scheduled to occur at mutually exclusive times. Thus, the agent would have to discard the possibility of achieving both goals. This leads to time being wasted by the assisting agents that could have been spent on different actions.

The execution framework can work with both heterogeneous agents (as presented in Chapter 5) and homogeneous agents (as presented in Chapter 6). However, the Robocup Rescue domain used in Chapter 5 has an explicit directionality about which agents request assistance and which agents provide help (police agents clear routes for ambulance agents). When working with homogeneous agents this distinction does not exist as agents are capable of both providing assistance and receiving assistance. This meant that the execution framework had to change how it allocates tasks to prevent dependency cycles. Should agents end up in a dependency cycle there exists no order in which agents can plan – as each agent is both waiting for the plan of one agent and providing its plan to another agent. The modification to the execution framework prevents dependency cycles by limiting agents to being members of one of three teams: an initial team and two other teams that an agent might be moved to, based on goal dependencies. The exact nature of the teams is described in Section 6.2.1. The limitation of three teams imposed by the execution framework is an implementation. Theoretically there could be as many teams as agents with complex dependencies – just so long as those dependencies are tree-like in structure with no cycles. By increasing the number of teams the execution framework

would be able to cope with more complex dependencies. For example, the Trucks domain used in Chapter 6 was limited to having goals to deliver packages across at most two networks (from one network directly to another network). With more teams the execution framework could create plans which involve packages crossing multiple networks.

By using dynamic teams (rather than arbitrarily splitting agents into predetermined teams) the execution framework is able to adapt to the specifics of a problem. For instance, in the Trucks domain agents are split between different networks and cannot move between them. If, however, the number of packages that are to be delivered from network A to network B is greater than the other way around then there should be more agents providing assistance in network A than in network B (as it will be the agents in network B that issue the subgoals for delivery of these packages). As the goal allocation algorithm proceeds, agents that have already been allocated goals will produce lower valued bids. This means that agents that have no allocated goals (and therefore still in the initial team) are more likely to be assigned goals. If the goal requires cooperation the agent is assigned to the team that has dependencies, if the goal was the subgoal of another goal then it is assigned to the team that provides assistance. In doing so agents will be more likely to be assigned to the team where they are most needed.

CHAPTER 8

FUTURE WORK

This chapter describes how some of the work presented in Chapters 3 to 6 could be extended and in some instances combined. Possible issues that could arise are also discussed.

Chapter 3 examined decisions concerned with how much of a limited time budget to allocate to planning. However, it did not consider the possibility of replanning. That is, an agent is given a goal, it then finds a plan, and finally executes the plan. This is because nothing in the domain necessitates replanning. In continual domains, external state changes or changes to the goals of an agent mean that, replanning is required. As such, an agent may switch back and forth between planning and execution multiple times. The ability to vary planning times allocated to planning for each planning phase, could lead to improvements in the amount of reward achieved. This is because the agent may be planning for a different number of goals, which would significantly impact the complexity of the problem instance. For example, in the UAV domain, the complexity of a problem instance is proportional to the factorial of the number of goals to achieve (when using brute force search). If replanning were to occur in the UAV domain the "loss limiting" MMS will spend less time planning. This is because goals are achieved through execution of the plan, meaning less goals are required to be achieved in future replanning. However, the planner is not aware of the limited time budget. Further work is needed to create a planner that is aware of this limited and decreasing time budget. This planner would not create plans that extend beyond an initial maximum makespan. The planner would also

reduce this maximum makespan as time passes during planning. This will help prevent the planner optimising parts of the plan that can never be executed because less of the time budget is available.

Chapter 4 looked at ways to increase total reward by acting whilst planning. In order to act whilst planning it is required that the amount of time allocated to planning be known at the start of planning. The work presented in Chapter 4 used a fixed planning time. However, a fixed time is not optimal for all problem instances. The combination of optimising the time allocated to planning with acting whilst planning should lead to further increases in reward. However, in continual domains, an agent will need to take into consideration the possibility of unforeseen state changes or the addition/removal of goals. These two issues can invalidate the state that the agent is planning for. In these circumstances, plans returned by planning may be invalid. As such, the agent is forced to abandon the current planning process and start again. This means that an agent must consider allocating a smaller amount of time to planning than would otherwise be optimal, so that the agent has an opportunity to act before the plan is invalidated. Future work could look at pre-empting replanning, and always planning whilst executing. As time passes, the "initial state" would be changed to reflect the current state. States that are no longer reachable would be pruned. If any new external state change occurs then any part of the plan space that is no longer valid because of this state changed pruned. If a new goal was added then the estimated reward from a given (planning) state reassessed to take into account the new goal.

Chapter 5 presented a way of using multi-agent planning in domains with shared time budgets. One of the problems with how tasks were assigned was that the more tasks that were in a problem instance, the more time was spent on allocating tasks.

It may be possible that an agent can attempt to achieve tasks that is has been allocated before allocation of all tasks has finished. In domains where tasks have deadlines, then tasks with earlier deadlines will be achieved earlier on in a plan. Using the same assumption that is made in Chapter 4 (that early parts of the plan will not change signif-

icantly) and if tasks are allocated in order of ascending deadlines, then agents can act in a way that is directed towards achieving these initial tasks and increase reward achieved. However, this is only true if the agent does not require any subtasks to be completed for it to achieve the task. For example, a police agent can start clearing blocked roads immediately as the clearing of roads has no preconditions that the agent cannot achieve itself. In contrast, a medic agent that requires a road to be unblocked to rescue a civilian cannot compute valid plans to rescue the civilian until it knows when the road will be cleared. Indeed it could be that no police agent is able to clear the road within the deadline, and so the road is never cleared. Clearly, one avenue for future work is to have different teams of agents plan with different initial states that reflect the earliest point at which they can begin executing.

In domains where an agent is oversubscribed, then it is possible for a given task to not be part of the optimal subset of tasks the agent is able to achieve. This presents a problem to acting before task allocation has finished. If an agent acts towards achieving a task before it can decide if that task is part of the optimal set of tasks it can achieve, then the agent will be committed to a sub-optimal task set. However, it was found in Chapter 4 that acting during planning increases the reward attained by an agent. This thesis argues that this will be the case with acting during task allocation, especially in situations where the time taken to allocate tasks dominates the time allocated to planning. This warrants further investigation of the AWP EF and how well it copes in domains that are oversubscribed.

When bidding for a task, an agent performs planning to see if it can achieve the task. However, the plan that is produced is a relaxed plan though and so may not be valid. If the relaxed plan does not generate any subtasks, however, then it is a valid plan. This means that an agent does not need to perform any planning, beyond what is necessary to bid for tasks, in order to act during task allocation. Future work could look into analysing the relaxed plan and, if it is valid in the non-relaxed domain, using it as a starting point to plan from. This would enable the agent to spend its planning time improving this valid

plan rather than recomputing it afresh.

A limitation of the implementation of the execution framework presented in Chapter 5 is that it can only work with two teams of agents. This is an artificial limitation that makes it easier to conceptualise about which team is producing subgoals that it needs achieved to complete its own goals, and which team is achieving those subgoals without producing subgoals of its own. Conceptually, there is nothing stopping the execution framework using more teams, just so long as cooperation between agents contains no cycles. That is, agents that have subgoals are dependent on the plans of the agents that will achieve those subgoals, and so long as there are no cycles of such dependencies there exists an order in which the agents may plan. For domains such as the Trucks domain, having multiple teams of agents would enable the delivery of packages across multiple networks. Future work would look into managing dynamic team formation. It would also try to reduce the maximum length of the chain of agents that are dependent on each other, which will help reduce the total amount of time spent planning. This thesis defines teams as groups of agents that independent of each other and so can plan in parallel. Thus, more teams means less agents executing in parallel, and instead executing serially – meaning more time is spent planning than might otherwise be necessary.

In each of the domains that the multi-agent execution framework was tested on, the ways in which agents might require cooperation was hand-coded. For instance, in the Robocup Rescue domain, ambulance agents cannot traverse blocked roads but police agents are capable of doing so. Thus, the execution framework for the Robocup Rescue domain uses a relaxed planning domain in which ambulance agents can traverse blocked roads. If a planner is capable of returning a plan from using this domain, then it can be inspected for move actions that traverse blocked edges. If such an action is found then it is used to create a new subgoal. However, there is no reason that the cooperation in the Robocup Rescue domain had to be hand-coded. If this cooperation could be automatically extracted from a domain, then the execution framework could be used more easily on other domains. All this information is in the problem domain and could be extracted. That

is, future work would analyse the domain for actions with preconditions that are not in the effects of any of the actions that the agent is capable of executing. If these predicates could be found in the effects of the actions of other agents then it is known that these two agent types might have to solve problems with goals that require cooperation. By detecting these actions it would be possible for future work to create the relaxed planning domain automatically, and without having to resort to hand-coding it.

8.0.1 Bringing it all together

There are two problems when trying to bring the work from the different research chapters together. Firstly, there are incompatibilities between how acting whilst planning and using online monitoring and control to optimise planning duration. To act whilst planning requires that it is possible to predict time and therefore state at the end of planning. Without being able to predict this, this information cannot be passed to the planner and it cannot reason about the effects of actions that are being executed during planning. Secondly, whilst it is possible to execute whilst acting in multi-agent domains used in Chapters 5 and 6, time taken to allocate tasks dominates planning time in complex problems. Thus, the benefit of acting whilst planning is significantly diminished. Multi-agent domains which have simple decompositions into single-agent planning problems would still be a significant benefit to acting whilst planning.

If agents do not act whilst planning then it would be possible for future work to use online monitoring to control planning times of each individual agent in the problem. In Chapters 5 and 6 we saw that the execution framework would plan for a fixed time for two separate sets of agents. However, not all agents had dependencies or had other agents depending on them. In such circumstances these agents could plan for the full allotment of time – a feature that was not implemented by the EF. But even within the set of agents that are directly cooperating it could be possible to alter the planning time. This is because all agents are planning for the state at which all agents will have finished planning – which is at a fixed point in time. That is, given two agents that are directly

cooperating and a with total planning time of 20 seconds, the execution frameworks would split this evenly between both agents. However, there is nothing to stop the first agent only planning for 5 seconds and letting the second agent plan for 15 seconds. This would be beneficial if the utility of planning was low for the first agent, but high for the second agent. The online monitoring and control execution framework presented in Chapter 3 could be used to monitor if early termination of planning for the first agent could be beneficial.

An issue that future work investigating using dynamic planning times would have to address is that, when agents are cooperating with multiple other agents it can be hard to know when to stop. That is, an agent might have a plan for all subgoals for one agent, but the plan might not have subgoals for a second agent yet. In some circumstances it will remain beneficial to plan and in others it might be better to stop planning and allow the dependent agents to plan. An example of the former is if the planning problems for the agents with subgoals are comparatively easy and would not benefit from additional planning time, but the additional planning time could lead to a plan that achieves all subgoals. In this case it would be better to allow the initial agent to continue to plan and delay the planning of the dependent agents. An example of the latter is the reverse, that continued planning does not lead to a plan that achieves all subgoals, but that the reduced planning time for the dependent agents means they fail to generate higher quality plans that they otherwise would have. In this situation early termination of planning of the initial agent would have been beneficial.

CHAPTER 9

CONCLUSION

This thesis examined how to effectively share time between planning and execution. It explored three sub-questions that look at different ways of sharing time between planning and execution.

9.1 Optimising Planning Durations in Continual Domains

This chapter addressed the research sub-question of "How long should an agent plan for in the presence of a fixed time budget". Specifically, it looked at how long to run an anytime planner for in a domain where limited time budgets are shared between planning and execution. It discussed two meta-management systems (MMS) that varied the amount of time allocated to planning, with the aim of maximising reward achieved by the end of the shared time budget.

Anytime planners produce a series of plans with monotonically increasing reward until the optimal plan is found. That is, further planning never produces a worse plan. However, in domains with a limited and shared time budget, planning has a cost. Planning consumes time, which reduces the amount of time available to execute actions, which in turn reduces the maximum reward achievable. This means that further planning is not always beneficial in domains with limited and shared time budgets.

The first MMS uses the outcome of planning on previous problem instances to predict the optimal time to allocate to planning. Key to this predictive MMS was the idea that the optimal planning time could be predicted from previous observations of the planning process on similar problems. That is, planning for problem instances with a similar complexity should have a similar cost/reward profile. Ultimately this MMS proved to be weak, as the cost/reward profile of a problem instance can vary substantially even within narrow bounds of differentiators of problem instances.

The work on the predictive MMS led to the development of an alternative loss limiting MMS that monitors the planning process online. This MMS decides to stop planning once the cost of time outstrips the reward increase from further planning. The cost of time is the loss of reward from not being able to complete as many goals due to a reduced time budget for execution. The loss limiting MMS was an improvement on the predictive MMS and fixed planning time MMSes. This thesis found that using an loss limiting MMS results in more reward than using a predictive MMS. However, a key problem with the loss limiting MMS is that since the time allocated to planning is not known in advance, then the planner does not know what the maximum makespan of a plan can be. This means further planning can end up improving parts of the plan that cannot be executed within the time budget.

The problems caused by an unknown planning time motivated work into how to increase reward when planning for a fixed time. Namely, how to act in a goal-directed way during planning.

9.2 Acting Whilst Planning During Continuous Execution

This chapter proposed several execution frameworks (EF) that can increase reward in domains with a shared time budget and unknown knowledge. The EFs make use of two contributions made by this chapter. The first contribution finds ways to act in a goal

directed way during planning. The second contribution is a way of modelling actions that are mid-execution that enables a larger proportion of planning and execution to occur concurrently than if just the first contribution was used. Each EF planned for a fixed duration using an anytime planner. These two contributions address the second sub-question of this thesis – "What should an agent do when planning?". This thesis finds that agents are capable of executing goal-directed actions with the use of heuristics, and that this increases reward.

Two techniques for selecting goal-directed actions were examined. The first technique assumes that, should replanning be necessary, any new plan generated will not differ significantly from the current plan. That is, the actions that would have been scheduled by the old plan to execute during the replanning phase are still worth executing. The second technique finds a new initial plan quickly, and uses this to schedule actions for execution during planning. Using a known planning time allows the EF to predict the actual state at the end of planning and use this as an initial state for planning. This allows the EF to inform the planner of state changes that are expected be occur by the end of planning. The second technique assumes that the initial plan produced will not differ significantly from the final plan produced by replanning, at least not in the early part of the plan. The second technique has the benefit that all actions in the plan are valid for the current knowledge state.

Modelling actions that are mid-execution allows planning with initial states that have actions in such a state. This means the end of planning does not need to be synchronised with a state in which no actions are executing. This allows replanning to start earlier and allows more time to be spent executing actions during replanning. Not being able to model actions mid-execution would mean either not executing actions that extend beyond the end of replanning, or waiting for actions to finish execution after the end of replanning. This would increase the time it takes to achieve the goal, and so decrease reward.

Both contributions were found to be effective at reducing the time it took to achieve the goal. However, the combination of both contributions was found to be highly effective.

9.3 Cooperation and Synchronisation in Multi-Agent Environments

This chapter contributed a way of improving cooperation and synchronisation between agents in multi-agent domains with time budgets shared between planning and execution. This addresses the research question about whether it is better for an execution framework to focus on generating plans quickly or generating high quality plans. It was found that in domains where planning time is shared between planning and executing generating plans quickly is more important than generating high quality plans. The execution framework uses a decentralised market-based allocation where agents bid for tasks. Where an agent cannot achieve a task by itself it can subcontract those parts of task. The agent was also able to specify a deadline by which a task was required.

The decentralised nature of task allocation and planning means that agents are able to find good plans quickly, and increase the global reward achieved. When tested in the Robocup Rescue domain, the algorithm was found to be very successful when compared to a single-agent planning approach. Beyond trivial problems, single-agent planning approaches (where a central agent makes all decisions) quickly become intractable. However, the algorithm sacrifices completeness in order to be able to find near-optimal solutions quickly. Specifically, the algorithm is unable to find solutions to an individual task that would benefit from multiple agents cooperating, but that this cooperation is not necessary. For example, given a task where the optimal plan is for two agents to execute two actions concurrently, the algorithm will produce a plan that uses one agent to execute the actions on in its own, one after the other.

9.4 Comparison of Execution Frameworks

This chapter compared the Acting Whilst Planning execution framework (AWP EF – from Chapter 4) and the Cooperation and Synchronisation execution framework (CS EF – from Chapter 5). It found that the CS EF is able solve more complex problems than the

AWP EF. Where domains require loose cooperation (eg. the Robocup Rescue and Trucks domains), the CS EF achieves more reward than the AWP EF. The reason for this is that the CS EF is able to break complex problems into a set of simpler single agent problems that can mostly be solved in parallel. However, where domains requires tight cooperation (eg. the Janitor domain) the AWP CS performs slightly better, obtaining slightly more reward. This is because of its use of a centralised planner that is able to better able to reason about global reward.

Appendices

APPENDIX A

JANITOR DOMAIN

```
(define (domain janitor)
    (:requirements :strips :fluents :durative-actions :adl
                :equality)

    (:predicates
        (node ?n)
        (edge ?n1 ?n2)
        (is-room ?n)
        (agent ?a)
        (at ?a ?n)
        (available ?a)
        (dirty ?rm)
        (extra-dirty ?rm)
        (cleaned ?rm)
    )

    (:functions
        (distance ?n1 ?n2)
        (dirtiness ?rm)
    )

    (:durative-action move
        :parameters (?a ?n1 ?n2)
        :duration (= ?duration (distance ?n1 ?n2))
        :condition (and (at start (at ?a ?n1))
                        (at start (edge ?n1 ?n2))
                        (at start (node ?n1))
                        (at start (node ?n2))
                        (at start (agent ?a)))
        :effect (and (at start (not (at ?a ?n1)))
                        (at end (at ?a ?n2)))
```

```
    )

    (:durative-action clean
        :parameters (?a ?rm)
        :duration (= ?duration (dirtiness ?rm))
        :condition (and (at start (dirty ?rm))
                        (over all (at ?a ?rm))
                        (at start (agent ?a))
                        (at start (available ?a))
                        (at start (is-room ?rm))
                        )
        :effect (and (at start (not (dirty ?rm)))
                        (at start (not (available ?a)))
                        (at end (cleaned ?rm))
                        (at end (available ?a))
                        )
    )

    (:durative-action extra-clean
        :parameters (?a1 ?a2 ?rm)
        :duration (= ?duration (dirtiness ?rm))
        :condition (and (at start (extra-dirty ?rm))
                        (at start (not (= ?a1 ?a2)))
                        (over all (at ?a1 ?rm))
                        (over all (at ?a2 ?rm))
                        (at start (agent ?a1))
                        (at start (agent ?a2))
                        (at start (available ?a1))
                        (at start (available ?a2))
                        (at start (is-room ?rm))
                        )
        :effect (and (at start (not (extra-dirty ?rm)))
                        (at end (cleaned ?rm))
                        (at start (not (available ?a1)))
                        (at start (not (available ?a2)))
                        (at end (available ?a1))
                        (at end (available ?a1))
                        )
    )
)
```

APPENDIX B

ROBOCUP RESCUE DOMAIN

```
(define (domain roborescue)
    (:requirements :strips :fluents :durative-actions
        :timed-initial-literals :adl :equality :typing
        :action-costs :preferences)
    (:types
        predicate - object
        moveable - object
        agent civilian - moveable
        police medic - agent
        node - object
        building hospital - node
    )

    (:predicates
        (available ?m - moveable)
        (at ?m - moveable ?n - node)
        (carrying ?m - medic ?c - civilian)
        (empty ?m - medic)
        (edge ?n1 ?n2 - node)
        (blocked-edge ?n1 ?n2 - node)
        (buried ?c - civilian)
        (unburied ?c - civilian)
        (alive ?c - civilian)
        (rescued ?c - civilian)
        (required ?p - predicate)
        (cleared ?n1 ?n2 - node ?p - predicate)
    )

    (:functions
        (buriedness ?c - civilian)
        (blockedness ?n1 ?n2 - node)
```

```
    (distance ?n1 ?n2 - node)
)

(:durative-action move
    :parameters (?a - agent ?n1 - node ?n2 - node)
    :duration ( = ?duration (distance ?n1 ?n2))
    :condition (and
        (at start (at ?a ?n1))
        (at start (edge ?n1 ?n2))
    )
    :effect (and
        (at start (not (at ?a ?n1)))
        (at end (at ?a ?n2))
    )
)

(:durative-action unblock
    :parameters (?a - police ?n1 - node ?n2 - node)
    :duration (= ?duration (blockedness ?n1 ?n2))
    :condition (and
        (over all (at ?a ?n1))
        (at start (available ?a))
        (at start (blocked-edge ?n1 ?n2))
    )
    :effect (and
        (at start (not (blocked-edge ?n1 ?n2)))
        (at start (not (blocked-edge ?n2 ?n1)))
        (at start (not (available ?a)))
        (at end (edge ?n1 ?n2))
        (at end (edge ?n2 ?n1))
        (at end (available ?a))
    )
)

(:durative-action load
    :parameters (?m - medic ?c - civilian ?b - building)
    :duration (= ?duration 30)
    :condition (and
        (over all (at ?m ?b))
        (at start (at ?c ?b))
        (at start (empty ?m))
        (at start (unburied ?c))
        (at start (alive ?c))
        (at start (available ?m))
    )
    :effect (and
```

```
        (at start (not (at ?c ?b)))
        (at start (not (empty ?m)))
        (at end (carrying ?m ?c))
        (at start (not (available ?m)))
        (at end (available ?m))
    )
)

(:durative-action unload
    :parameters (?m - medic ?c - civilian ?b - hospital)
    :duration (= ?duration 30)
    :condition (and
        (over all (at ?m ?b))
        (at start (carrying ?m ?c))
        (over all (alive ?c))
        (at start (available ?m))
    )
    :effect (and
        (at start (not (carrying ?m ?c)))
        (at end (at ?c ?b))
        (at end (empty ?m))
        (at end (rescued ?c))
        (at start (not (available ?m)))
        (at end (available ?m))
    )
)

(:durative-action rescue
    :parameters (?m - medic ?c - civilian ?b - building)
    :duration (= ?duration (buriedness ?c))
    :condition (and
        (over all (at ?m ?b))
        (at start (at ?c ?b))
        (at start (buried ?c))
        (at start (alive ?c))
        (at start (available ?m))
    )
    :effect (and
        (at start (not (buried ?c)))
        (at end (unburied ?c))
        (at start (not (available ?m)))
        (at end (available ?m))
    )
)

(:durative-action clear
```

```
        :parameters (?n1 - node ?n2 - node ?p - predicate)
        :duration (= ?duration 0)
        :condition (and
            (at start (edge ?n1 ?n2))
            (at start (required ?p))
        )
        :effect (and
            (at start (cleared ?n1 ?n2 ?p))
        )
    )
)
```

APPENDIX C

TRUCKS DOMAIN

```
; IPC5 Domain: Trucks Time-TIL
; Authors: Yannis Dimopoulos, Alfonso Gerevini and Alessandro
; Saetti

(define (domain trucks-and-boats)
(:requirements :typing :adl :durative-actions :fluents
    :timed-initial-literals)

(:types
    vehiclearea location locatable - object
    vehicle package - locatable
    truck boat - vehicle
)

(:predicates
    (at ?x - locatable ?l - location)
    (in ?p - package ?v - vehicle ?a - vehiclearea)
    (connected-by-land ?x ?y - location)
    (connected-by-sea ?x ?y - location)
    (free ?a - vehiclearea ?v - vehicle)
    (delivered ?p - package ?l - location)
    (at-destination ?p - package ?l - location)
    (closer ?a1 - vehiclearea ?a2 - vehiclearea)
    (deliverable ?p - package ?l - location)
)

(:functions (travel-time ?from ?to - location))

(:durative-action load
    :parameters (?p - package ?v - vehicle ?a1 - vehiclearea
                 ?l - location)
```

```
    :duration (= ?duration 1)
    :condition (and
        (at start (at ?p ?l))
        (at start (free ?a1 ?v))
        (at start
            (forall (?a2 - vehiclearea)
                (imply (closer ?a2 ?a1) (free ?a2 ?v))
            )
        )
        (over all (at ?v ?l))
        (over all
            (forall (?a2 - vehiclearea)
                (imply (closer ?a2 ?a1) (free ?a2 ?v))
            )
        )
    )
    :effect (and
        (at start (not (at ?p ?l)))
        (at start (not (free ?a1 ?v)))
        (at end (in ?p ?v ?a1))
    )
)

(:durative-action unload
    :parameters (?p - package ?v - vehicle ?a1 - vehiclearea
                 ?l - location)
    :duration (= ?duration 1)
    :condition (and
        (at start (in ?p ?v ?a1))
        (at start
            (forall (?a2 - vehiclearea)
                (imply (closer ?a2 ?a1) (free ?a2 ?v))
            )
        )
        (over all (at ?v ?l))
        (over all
            (forall (?a2 - vehiclearea)
                (imply (closer ?a2 ?a1) (free ?a2 ?v))
            )
        )
    )
    :effect (and
        (at start (not (in ?p ?v ?a1)))
        (at end (free ?a1 ?v))
        (at end (at ?p ?l))
    )
```

```
)

(:durative-action drive
    :parameters (?t - truck ?from ?to - location)
    :duration (= ?duration (travel-time ?from ?to))
    :condition (and
        (at start (at ?t ?from))
        (over all (connected-by-land ?from ?to))
    )
    :effect (and
        (at start (not (at ?t ?from)))
        (at end (at ?t ?to))
    )
)

(:durative-action sail
    :parameters (?b - boat ?from ?to - location)
    :duration (= ?duration (travel-time ?from ?to))
    :condition (and
        (at start (at ?b ?from))
        (over all (connected-by-sea ?from ?to))
    )
    :effect (and
        (at start (not (at ?b ?from)))
        (at end (at ?b ?to))
    )
)

(:durative-action deliver-ontime
    :parameters (?p - package ?l - location)
    :duration (= ?duration 1)
    :condition (and
        (over all (at ?p ?l))
        (at end (deliverable ?p ?l))
    )
    :effect (and
        (at end (not (at ?p ?l)))
        (at end (delivered ?p ?l))
        (at end (at-destination ?p ?l))
    )
)

(:durative-action deliver-anytime
    :parameters (?p - package ?l - location)
    :duration (= ?duration 1)
    :condition (and
```

```
        (at start (at ?p ?l))
        (over all (at ?p ?l))
    )
    :effect (and
        (at end (not (at ?p ?l)))
        (at end (at-destination ?p ?l)))
    )
)
```

BIBLIOGRAPHY

[1] Martin Andersson and Tuomas Sandholm. Contract type sequencing for reallocative negotiation. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 154–160, 2000.

[2] Jeremy W. Baxter, Jack Hargreaves, Nick Hawes, and Rustam Stolkin. Controlling anytime scheduling of observation tasks. In *Proceedings of AI-2012, The Thirty-Second SGAI International Conference on Artificial Intelligence*, December 2012.

[3] J. Benton, A. J. Coles, and A. I. Coles. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the Twenty Second International Conference on Automated Planning and Scheduling (ICAPS-12)*, June 2012.

[4] Ronen I Brafman and Carmel Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, pages 28–35, 2008.

[5] Michael Brenner. Multiagent planning with partially ordered temporal plans. In *IJCAI*, volume 3, pages 1513–1514, 2003.

[6] Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.

[7] Arne Brutschy, Giovanni Pini, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Self-organized task allocation to sequentially interdependent tasks in swarm robotics. *Autonomous agents and multi-agent systems*, 28(1):101–125, 2014.

[8] Ethan Burns, J Benton, Wheeler Ruml, Sung Wook Yoon, and Minh Binh Do. Anticipatory on-line planning. In *ICAPS*, 2012.

[9] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *ICAPS*, pages 42–49, 2010.

[10] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In *AAAI*, pages 892–897, 2008.

[11] International Planning Competition. Benchmark Domains and Problems of IPC-5. `http://ipc06.icaps-conference.org/deterministic/domains.html`, 2006. [Online; accessed 2018-02-07].

[12] Matt Crosby and Michael Rovatsos. Heuristic multiagent planning with self-interested agents. In *The 10th International Conference on Autonomous Agents and Multiagent Systems*, volume 3, pages 1213–1214, 2011.

[13] Matthew Crosby, Anders Jonsson, and Michael Rovatsos. A single-agent approach to multiagent planning. In *21st European Conference on Artificial Intelligence (ECAI)*, pages 237–242, 2014.

[14] Matthew Crosby, Michael Rovatsos, and Ronald P.A. Petrick. Automated agent decomposition for classical planning. In *ICAPS*, pages 46–54, 2013.

[15] William Cushing, J Benton, and Subbarao Kambhampati. Replanning as a deliberative re-selection of objectives. *Arizona State University CSE Department TR*, 2008.

[16] William Cushing, Subbarao Kambhampati, Daniel S Weld, et al. When is temporal planning really temporal? In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 1852–1859. Morgan Kaufmann Publishers Inc., 2007.

[17] M Bernardine Dias. *Traderbots: A new paradigm for robust and efficient multirobot coordination in dynamic environments*. PhD thesis, Carnegie Mellon University Pittsburgh, 2004.

[18] M Bernardine Dias, Robert Zlot, Nidhi Kalra, and Anthony Stentz. Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, 94(7):1257–1270, 2006.

[19] Minh Do and Subbarao Kambhampati. Sapa: A domain-independent heuristic metric temporal planner. In *Sixth European Conference on Planning*, 2014.

[20] Minh Binh Do and Subbarao Kambhampati. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)*, 20:155–194, 2003.

[21] S Edelkamp and J Hoffmann. PDDL 2.2: The language for the classical part of the 4th international planning competition, albert ludwigs universität institüt fur informatik, freiburg. Technical report, Germany, Technical Report, 2004.

[22] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[23] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003.

[24] Christian Fritz and Sheila A McIlraith. Monitoring plan optimality during execution. In *ICAPS*, pages 144–151, 2007.

[25] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3. *The Language of the Fifth International Planning Competition. Tech. Rep. Technical Report, Department of Electronics for Automation, University of Brescia, Italy*, 75, 2005.

[26] Alfonso E Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5):619–668, 2009.

[27] Brian P Gerkey and Maja J Matarić. Sold!: Auction methods for multirobot coordination. *IEEE transactions on robotics and automation*, 18(5):758–768, 2002.

[28] Brian P Gerkey and Maja J Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, 2004.

[29] Eric A Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 28:267–297, 2007.

[30] Eric A Hansen and Shlomo Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1):139–157, 2001.

[31] Jack Hargreaves and Nick Hawes. What to do whilst replanning? In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2014), UK*, 2014.

[32] Patrik Haslum and Hector Geffner. Heuristic planning with time and resources. In *Sixth European Conference on Planning*, 2014.

[33] M Helmert. Changes in PDDL 3.1. `http://ipc.informatik.unifreiburg.de/PddlExtension`, 2008. Unpublished summary from the IPC-2008 website.

[34] Jörg Hoffmann and Ronen Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS*, volume 2005, 2005.

[35] Jörg Hoffmann and Ronen I Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6):507–541, 2006.

[36] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[37] Anders Jonsson and Michael Rovatsos. Scaling up multiagent planning: A best-response approach. In *In Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011)*, pages 114–121, 2011.

[38] Nidhi Kalra and Alcherio Martinoli. Comparative study of market-based and threshold-based task allocation. In *Distributed Autonomous Robotic Systems*, volume 7, pages 91–101. Springer, 2006.

[39] Hiroaki Kitano and Satoshi Tadokoro. Robocup rescue: A grand challenge for multiagent and intelligent systems. *AI magazine*, 22(1):39, 2001.

[40] Sven Koenig and Maxim Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[41] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1):93–146, 2004.

[42] G Ayorkor Korsah, Anthony Stentz, and M Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013.

[43] Sarit Kraus. Automated negotiation and decision making in multiagent environments. In *Multi-agent Systems and Applications*, pages 150–172. Springer, 2001.

[44] Seth Lemons, J Benton, Wheeler Ruml, Minh Binh Do, and Sung Wook Yoon. Continual on-line planning as decision-theoretic incremental heuristic search. In *AAAI Spring Symposium: Embedded Reasoning*, 2010.

[45] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[46] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, page None, 2003.

[47] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL-the planning domain definition language*, 1998.

[48] Romeo Sanchez Nigenda and Subbarao Kambhampati. Planning graph heuristics for selecting objectives in over-subscription planning problems. In *ICAPS*, pages 192–201, 2005.

[49] Raz Nissim, Ronen I Brafman, and Carmel Domshlak. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 1323–1330, 2010.

[50] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley Publishing, Reading, MA, 1984.

[51] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3-4):193–204, 1970.

[52] Martha E Pollack and John F Horty. There's more to life than making plans: plan management in dynamic, multiagent environments. *AI Magazine*, 20(4):71, 1999.

[53] Silvia Richter, Jordan Tyler Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In *ICAPS*, pages 137–144, 2010.

[54] Stuart Jonathan Russell and Eric Wefald. *Do the right thing: studies in limited rationality.* MIT press, 1991.

[55] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1):1–54, 2002.

[56] Tuomas W Sandholm. Limitations of the Vickrey auction in computational multiagent systems. In *Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96)*, pages 299–306, 1996.

[57] David E Smith. Choosing objectives in over-subscription planning. In *ICAPS*, volume 4, pages 393–=401, 2004.

[58] Anthony (Tony) Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, pages 3310–3317, May 1994.

[59] Jordan Tyler Thayer and Wheeler Ruml. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *ICAPS*, pages 355–362, 2008.

[60] Menkes Van Den Briel, Romeo Sanchez, Minh Binh Do, and Subbarao Kambhampati. Effective approaches for partial satisfaction (over-subscription) planning. In *AAAI*, pages 562–569, 2004.

[61] Peter R Wurman, Michael P Wellman, and William E Walsh. The michigan internet auctionbot: A configurable auction server for human and software agents. In *Proceedings of the second international conference on Autonomous agents*, pages 301–308. ACM, 1998.

[62] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. Multi-robot decentralized exploration using a trade-based approach. In *International Conference on Informatics in Control, Automation and Robotics*, pages 99–105, 2011.

[63] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10, 2013.

[64] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, pages 352–359, 2007.

[65] Sung Wook Yoon, Alan Fern, Robert Givan, and Subbarao Kambhampati. Probabilistic planning via determinization in hindsight. In *AAAI*, pages 1010–1016, 2008.

[66] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73–83, 1996.

[67] Robert Zlot and Anthony Stentz. Market-based multirobot coordination for complex tasks. *The International Journal of Robotics Research*, 25(1):73–101, 2006.